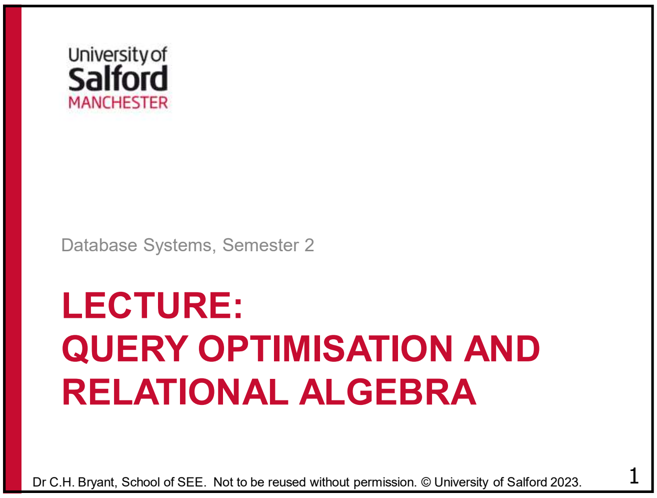
University of Salford MANCHESTER

Database Systems Lecture Slides

CRN 32741, UMC G400 10045

Dr Bryant

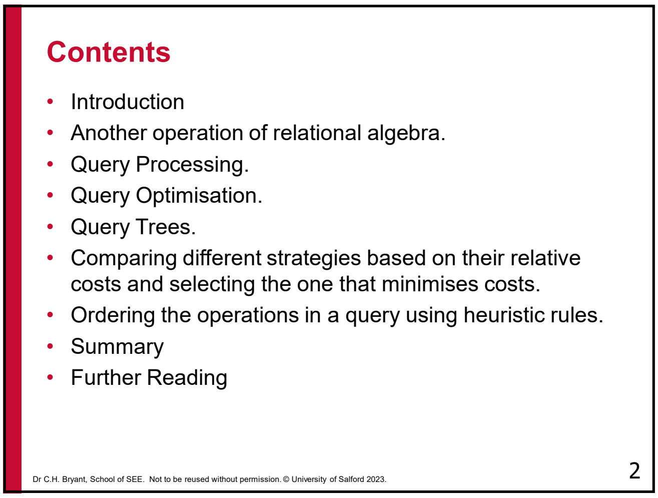
Semester 2 of 2023/24



University of Salford MANCHESTER

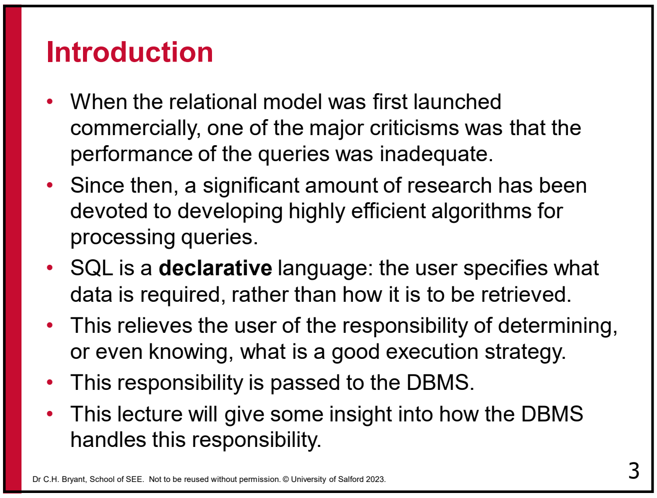
Database Systems, Semester 2

LECTURE: QUERY OPTIMISATION AND RELATIONAL ALGEBRA



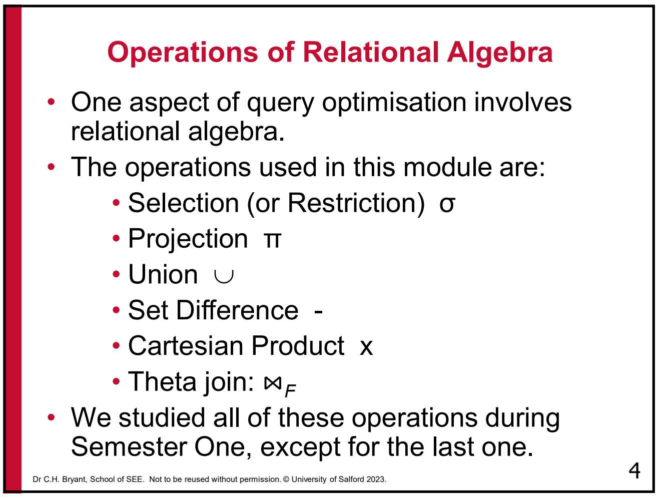
Contents

- Introduction
- Another operation of relational algebra.
- Query Processing.
- Query Optimisation.
- Query Trees.
- Comparing different strategies based on their relative costs and selecting the one that minimises costs.
- Ordering the operations in a query using heuristic rules.
- Summary
- Further Reading



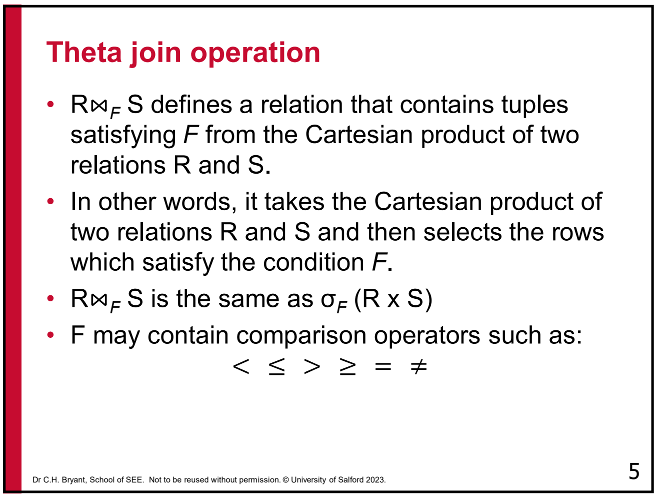
Introduction

- When the relational model was first launched commercially, one of the major criticisms was that the performance of the queries was inadequate.
- Since then, a significant amount of research has been devoted to developing highly efficient algorithms for processing queries.
- SQL is a **declarative** language: the user specifies what data is required, rather than how it is to be retrieved.
- This relieves the user of the responsibility of determining, or even knowing, what is a good execution strategy.
- This responsibility is passed to the DBMS.
- This lecture will give some insight into how the DBMS handles this responsibility.



Operations of Relational Algebra

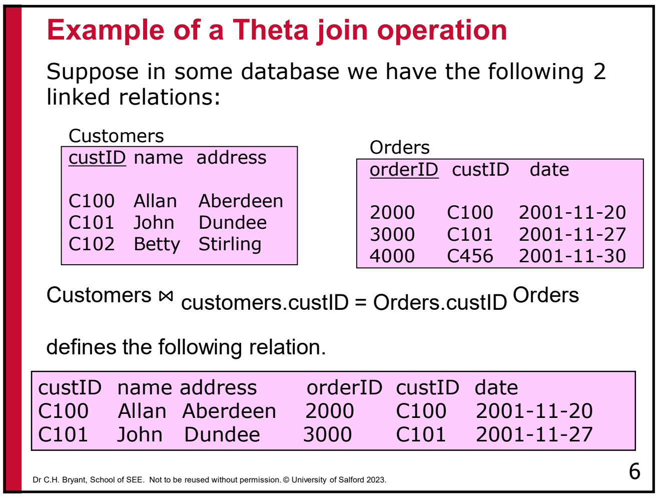
- One aspect of query optimisation involves relational algebra.
- The operations used in this module are:
  - Selection (or Restriction) σ
  - Projection π
  - Union $\cup$
  - Set Difference -
  - Cartesian Product x
  - Theta join: $\bowtie_F$
- We studied all of these operations during Semester One, except for the last one.



Theta join operation

- $R \bowtie_F S$ defines a relation that contains tuples satisfying *F* from the Cartesian product of two relations R and S.
- In other words, it takes the Cartesian product of two relations R and S and then selects the rows which satisfy the condition *F*.
- $R \bowtie_F S$ is the same as $\sigma_F (R \times S)$
- F may contain comparison operators such as:

$$< \ \leq \ > \ \geq \ = \ \neq$$



Example of a Theta join operation

Suppose in some database we have the following 2 linked relations:

Customers

custID	name	address
C100	Allan	Aberdeen
C101	John	Dundee
C102	Betty	Stirling

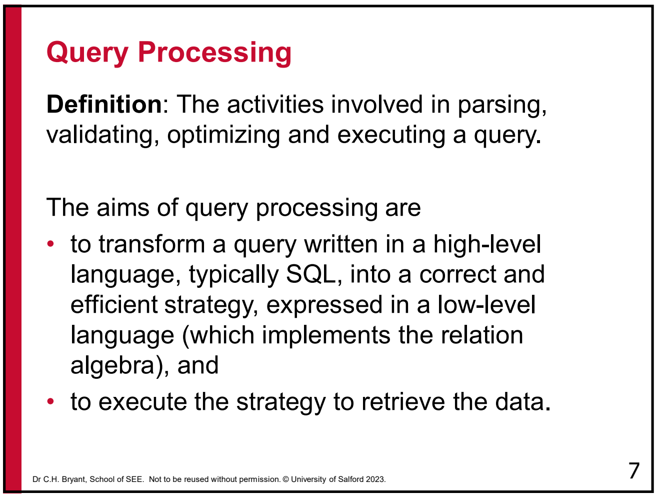
Orders

orderID	custID	date
2000	C100	2001-11-20
3000	C101	2001-11-27
4000	C456	2001-11-30

Customers $\bowtie_{\text{customers.custID = Orders.custID}}$ Orders

defines the following relation.

custID	name	address	orderID	custID	date
C100	Allan	Aberdeen	2000	C100	2001-11-20
C101	John	Dundee	3000	C101	2001-11-27

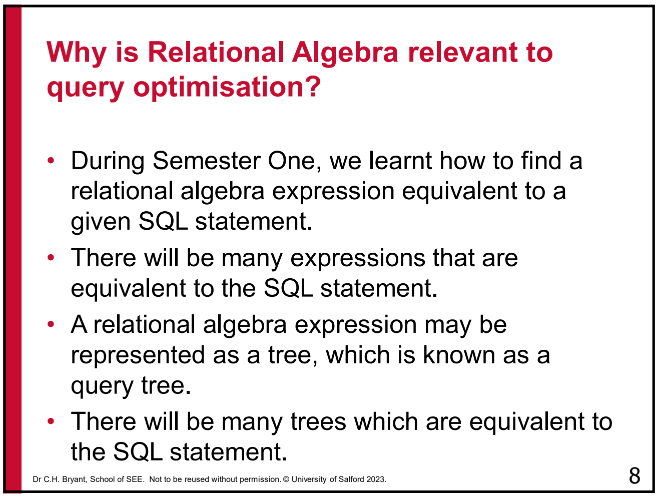


Query Processing

Definition: The activities involved in parsing, validating, optimizing and executing a query.

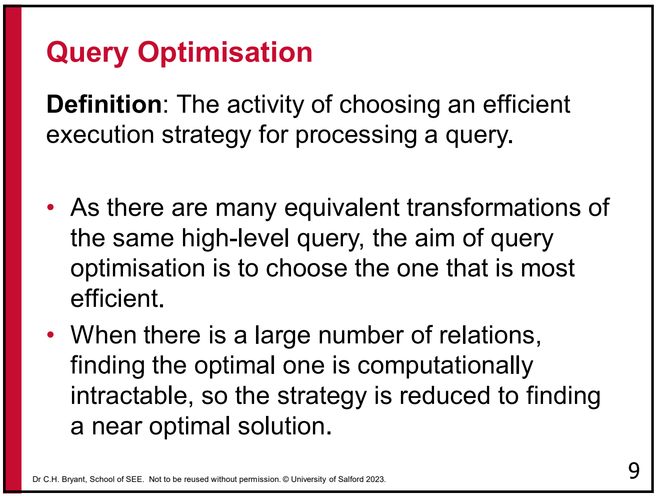
The aims of query processing are

- to transform a query written in a high-level language, typically SQL, into a correct and efficient strategy, expressed in a low-level language (which implements the relation algebra), and
- to execute the strategy to retrieve the data.



Why is Relational Algebra relevant to query optimisation?

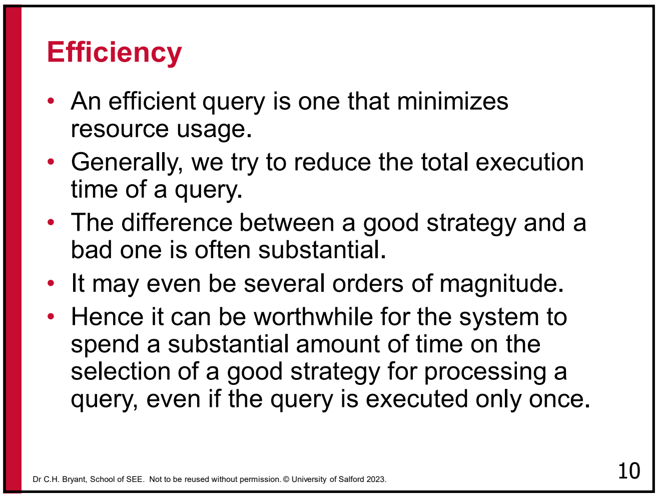
- During Semester One, we learnt how to find a relational algebra expression equivalent to a given SQL statement.
- There will be many expressions that are equivalent to the SQL statement.
- A relational algebra expression may be represented as a tree, which is known as a query tree.
- There will be many trees which are equivalent to the SQL statement.



Query Optimisation

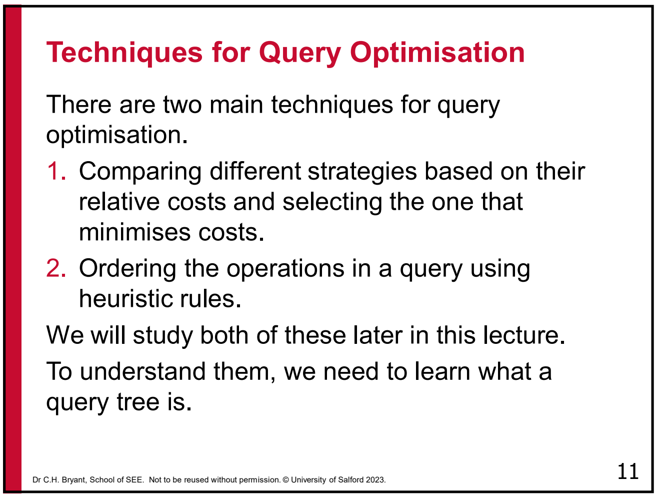
Definition: The activity of choosing an efficient execution strategy for processing a query.

- As there are many equivalent transformations of the same high-level query, the aim of query optimisation is to choose the one that is most efficient.
- When there is a large number of relations, finding the optimal one is computationally intractable, so the strategy is reduced to finding a near optimal solution.



Efficiency

- An efficient query is one that minimizes resource usage.
- Generally, we try to reduce the total execution time of a query.
- The difference between a good strategy and a bad one is often substantial.
- It may even be several orders of magnitude.
- Hence it can be worthwhile for the system to spend a substantial amount of time on the selection of a good strategy for processing a query, even if the query is executed only once.



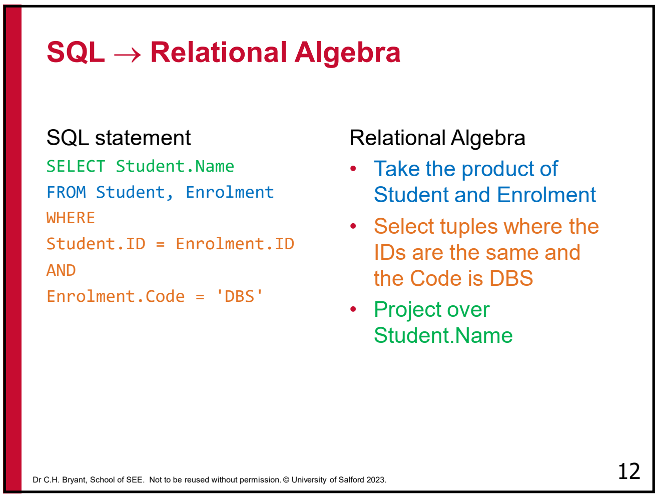
Techniques for Query Optimisation

There are two main techniques for query optimisation.

1. Comparing different strategies based on their relative costs and selecting the one that minimises costs.
2. Ordering the operations in a query using heuristic rules.

We will study both of these later in this lecture.

To understand them, we need to learn what a query tree is.



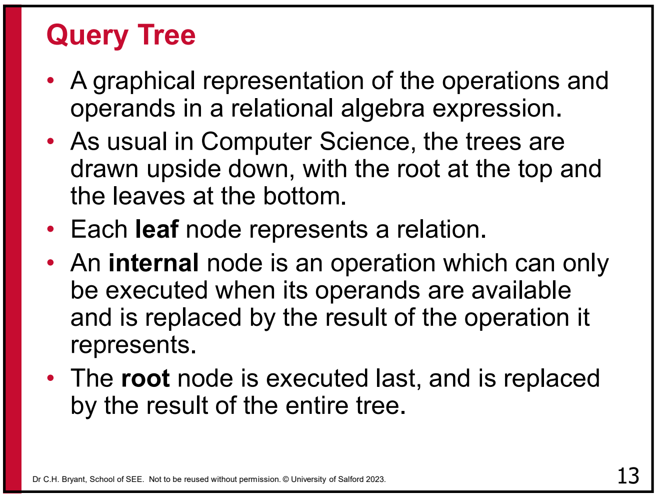
SQL → Relational Algebra

SQL statement

```
SELECT Student.Name
FROM Student, Enrolment
WHERE
Student.ID = Enrolment.ID
AND
Enrolment.Code = 'DBS'
```

Relational Algebra

- Take the product of Student and Enrolment
- Select tuples where the IDs are the same and the Code is DBS
- Project over Student.Name

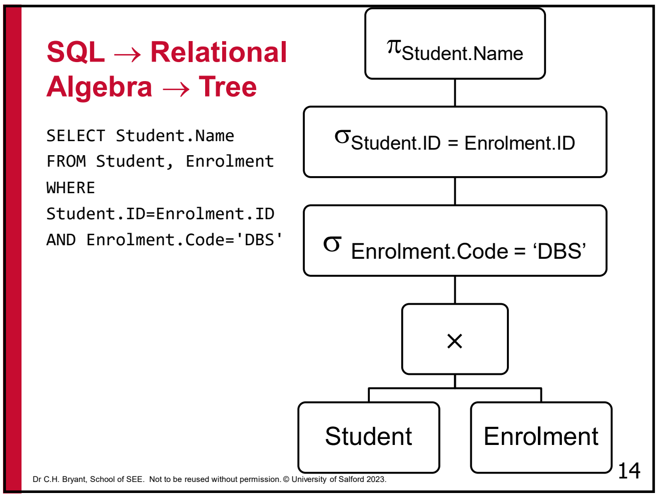


Query Tree

- A graphical representation of the operations and operands in a relational algebra expression.
- As usual in Computer Science, the trees are drawn upside down, with the root at the top and the leaves at the bottom.
- Each **leaf** node represents a relation.
- An **internal** node is an operation which can only be executed when its operands are available and is replaced by the result of the operation it represents.
- The **root** node is executed last, and is replaced by the result of the entire tree.

13

13



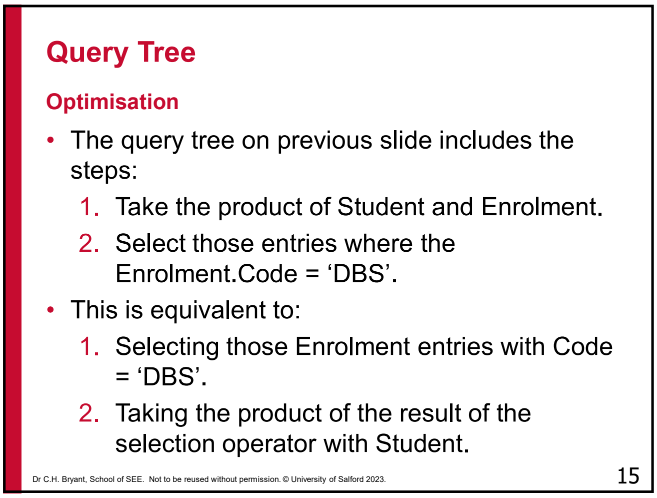
SQL → Relational Algebra → Tree

```
SELECT Student.Name
FROM Student, Enrolment
WHERE
Student.ID=Enrolment.ID
AND Enrolment.Code='DBS'
```

$\pi_{Student.Name}$
$\sigma_{Student.ID\ =\ Enrolment.ID}$
σ Enrolment.Code = ‘DBS’
×
Student
Enrolment

14

14



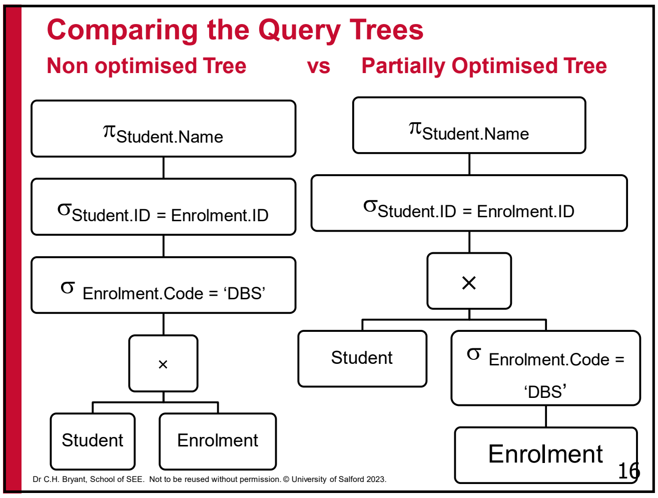
Query Tree

Optimisation

- The query tree on previous slide includes the steps:
  1. Take the product of Student and Enrolment.
  2. Select those entries where the Enrolment.Code = ‘DBS’.
- This is equivalent to:
  1. Selecting those Enrolment entries with Code = ‘DBS’.
  2. Taking the product of the result of the selection operator with Student.

15

15



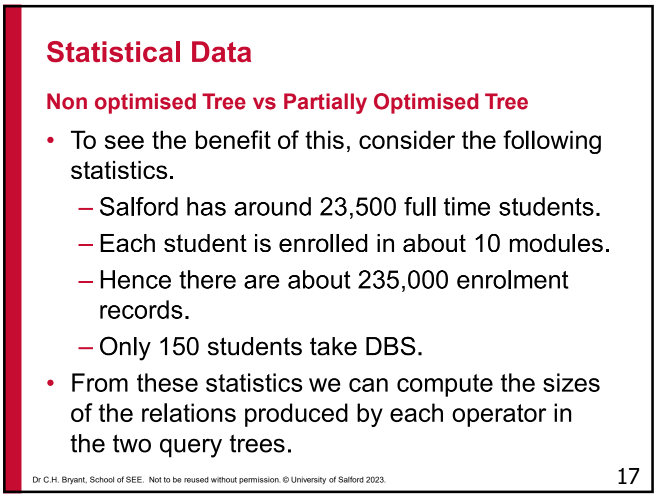
Comparing the Query Trees

Non optimised Tree vs Partially Optimised Tree

$\pi_{Student.Name}$
$\sigma_{Student.ID\ =\ Enrolment.ID}$
σ Enrolment.Code = ‘DBS’
×
Student
Enrolment
$\pi_{Student.Name}$
$\sigma_{Student.ID\ =\ Enrolment.ID}$
×
Student
σ Enrolment.Code = ‘DBS’
Enrolment

16

16



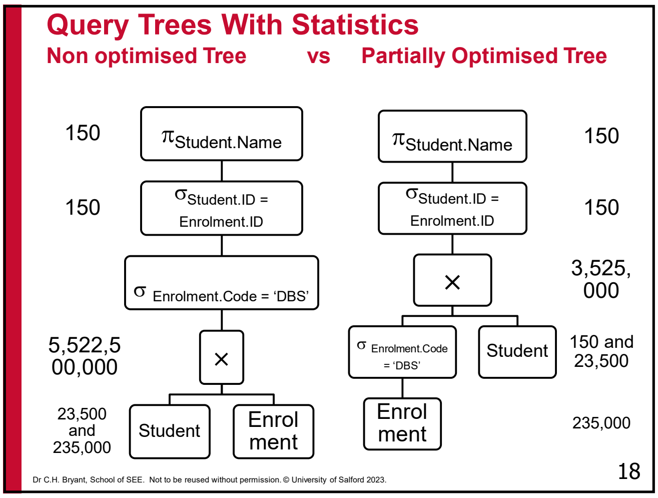
Statistical Data

Non optimised Tree vs Partially Optimised Tree

- To see the benefit of this, consider the following statistics.
  - Salford has around 23,500 full time students.
  - Each student is enrolled in about 10 modules.
  - Hence there are about 235,000 enrolment records.
  - Only 150 students take DBS.
- From these statistics we can compute the sizes of the relations produced by each operator in the two query trees.

17

17

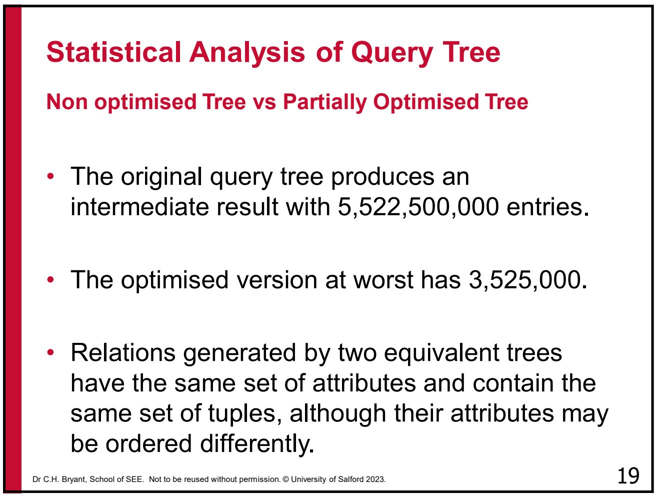


Query Trees With Statistics

Non optimised Tree vs Partially Optimised Tree

18

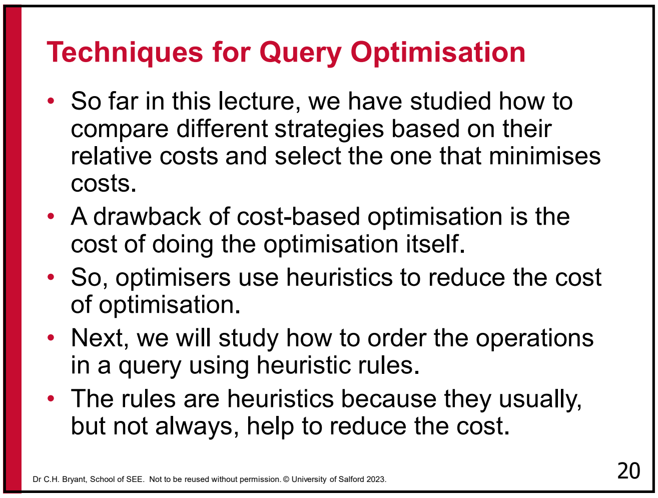
18



Statistical Analysis of Query Tree

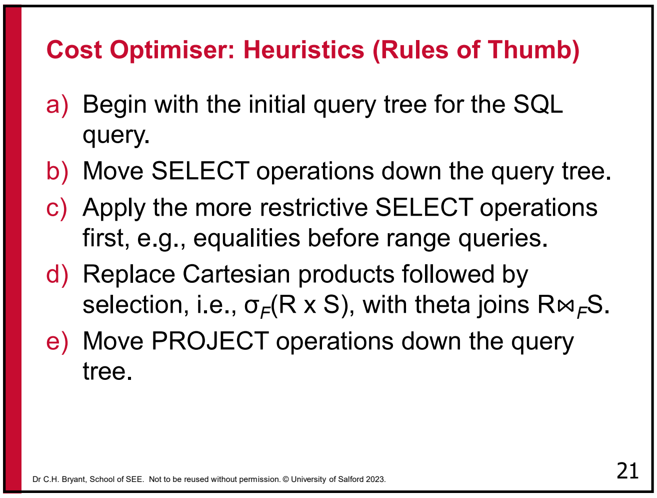
Non optimised Tree vs Partially Optimised Tree

- The original query tree produces an intermediate result with 5,522,500,000 entries.
- The optimised version at worst has 3,525,000.
- Relations generated by two equivalent trees have the same set of attributes and contain the same set of tuples, although their attributes may be ordered differently.



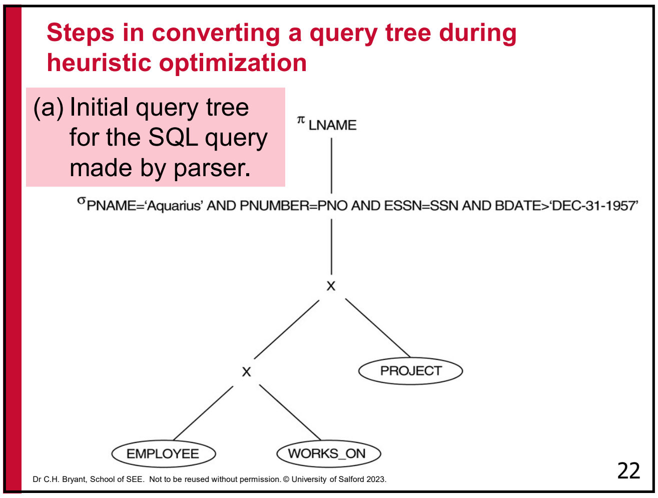
Techniques for Query Optimisation

- So far in this lecture, we have studied how to compare different strategies based on their relative costs and select the one that minimises costs.
- A drawback of cost-based optimisation is the cost of doing the optimisation itself.
- So, optimisers use heuristics to reduce the cost of optimisation.
- Next, we will study how to order the operations in a query using heuristic rules.
- The rules are heuristics because they usually, but not always, help to reduce the cost.



Cost Optimiser: Heuristics (Rules of Thumb)

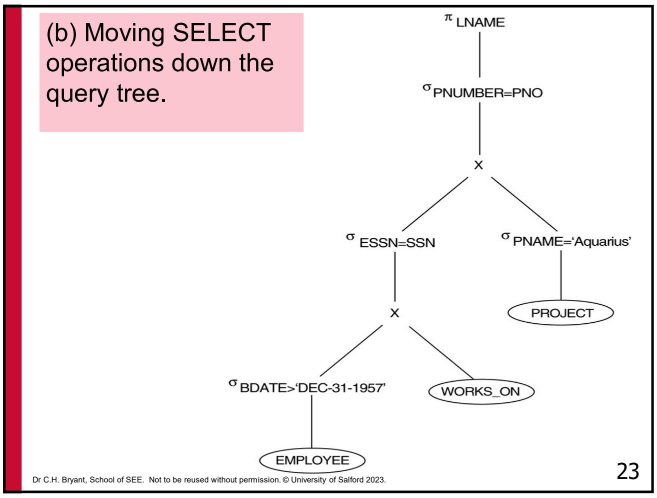
a) Begin with the initial query tree for the SQL query.
b) Move SELECT operations down the query tree.
c) Apply the more restrictive SELECT operations first, e.g., equalities before range queries.
d) Replace Cartesian products followed by selection, i.e., $\sigma_F(R \times S)$, with theta joins $R \bowtie_F S$.
e) Move PROJECT operations down the query tree.



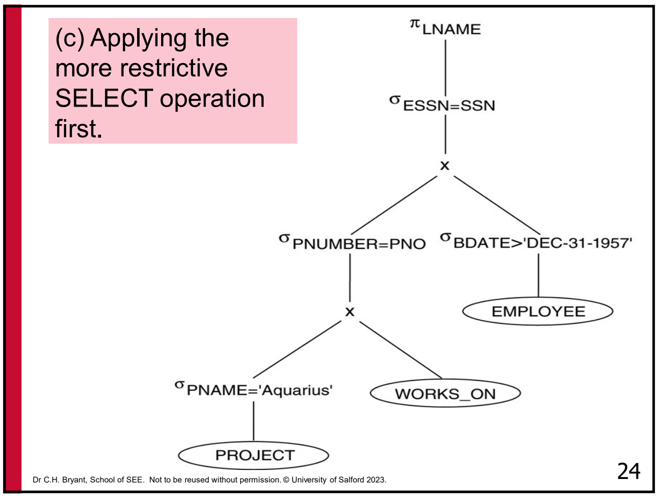
Steps in converting a query tree during heuristic optimization

(a) Initial query tree for the SQL query made by parser.

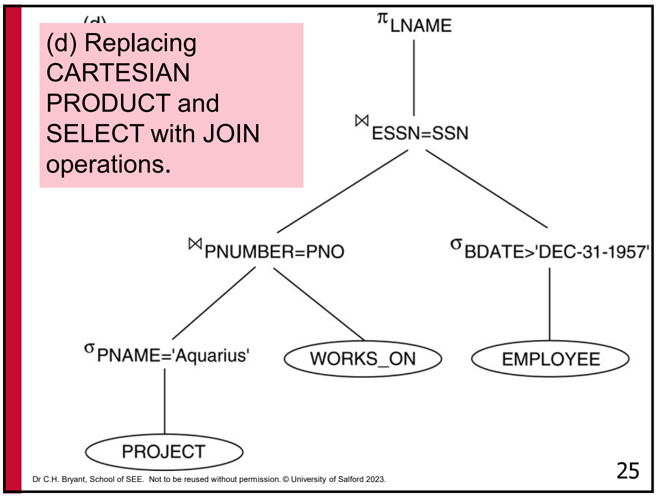
π LNAME
σ PNAME='Aquarius' AND PNUMBER=PNO AND ESSN=SSN AND BDATE>'DEC-31-1957'
X
X
PROJECT
EMPLOYEE
WORKS_ON



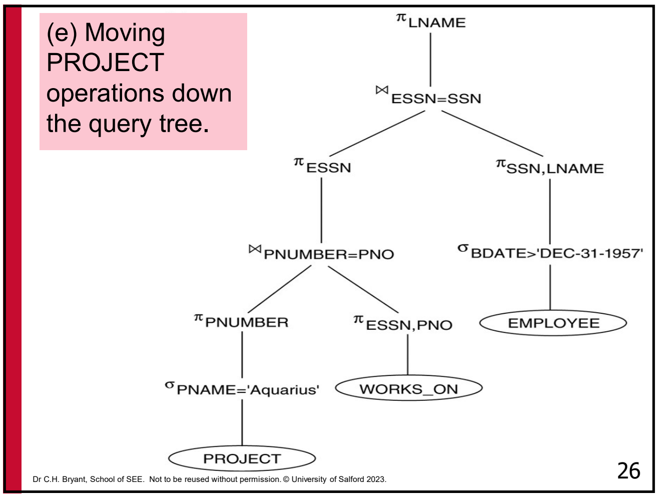
(b) Moving SELECT operations down the query tree.



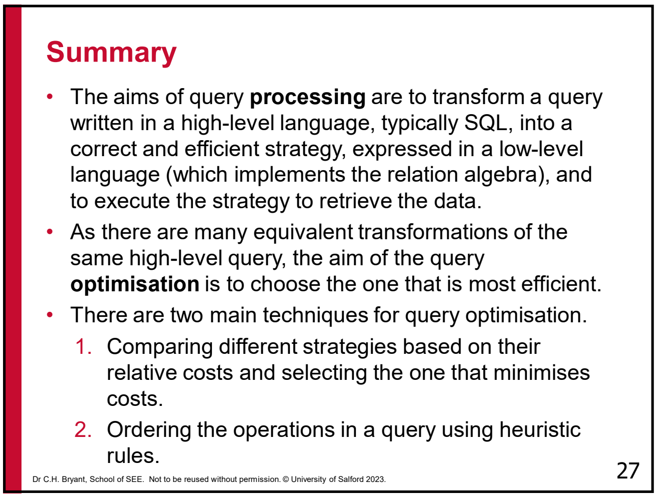
(c) Applying the more restrictive SELECT operation first.



(d) Replacing CARTESIAN PRODUCT and SELECT with JOIN operations.

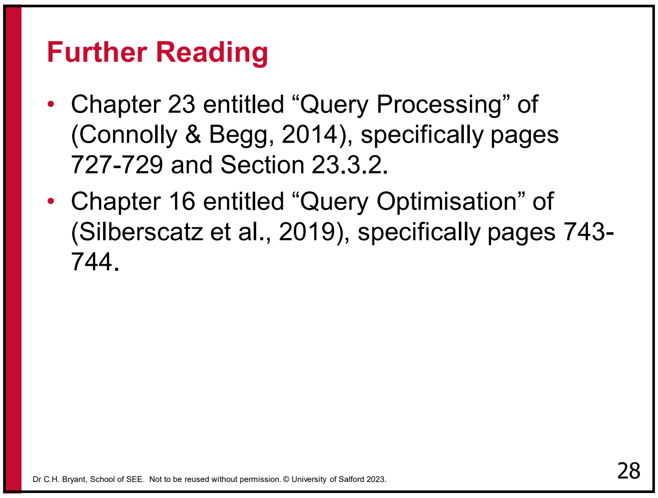


(e) Moving PROJECT operations down the query tree.



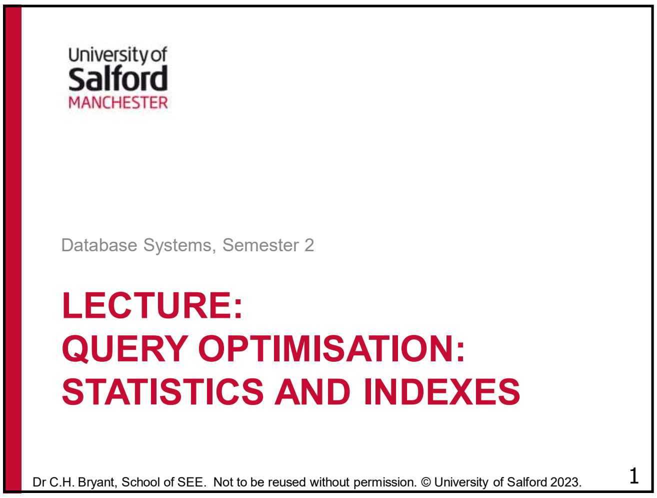
Summary

- The aims of query **processing** are to transform a query written in a high-level language, typically SQL, into a correct and efficient strategy, expressed in a low-level language (which implements the relation algebra), and to execute the strategy to retrieve the data.
- As there are many equivalent transformations of the same high-level query, the aim of the query **optimisation** is to choose the one that is most efficient.
- There are two main techniques for query optimisation.
  1. Comparing different strategies based on their relative costs and selecting the one that minimises costs.
  2. Ordering the operations in a query using heuristic rules.



Further Reading

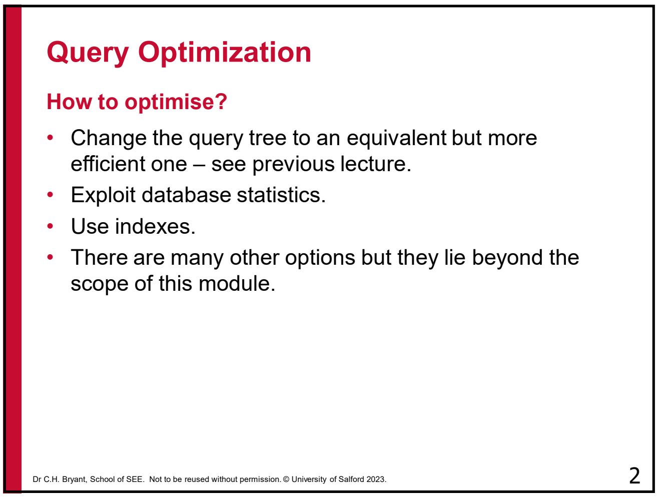
- Chapter 23 entitled “Query Processing” of (Connolly & Begg, 2014), specifically pages 727-729 and Section 23.3.2.
- Chapter 16 entitled “Query Optimisation” of (Silberscatz et al., 2019), specifically pages 743-744.



University of Salford MANCHESTER

Database Systems, Semester 2

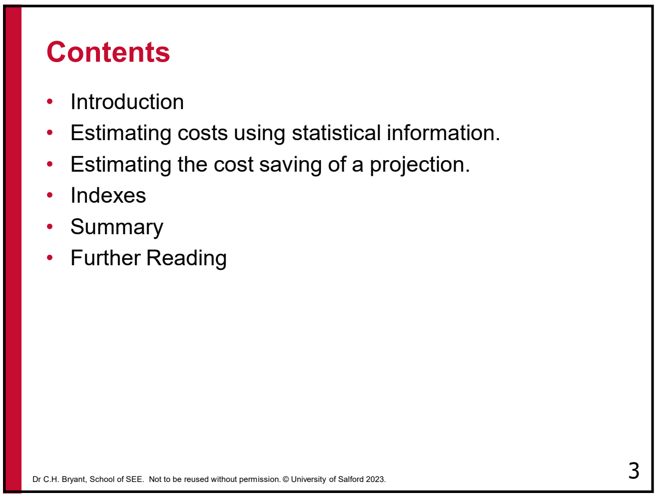
LECTURE: QUERY OPTIMISATION: STATISTICS AND INDEXES



Query Optimization

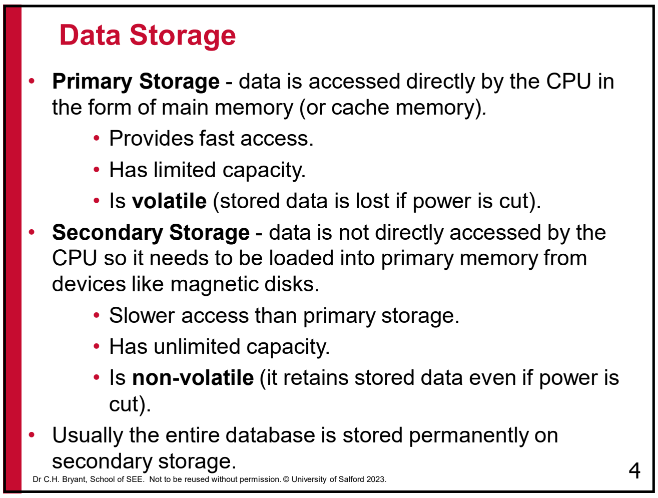
How to optimise?

- Change the query tree to an equivalent but more efficient one – see previous lecture.
- Exploit database statistics.
- Use indexes.
- There are many other options but they lie beyond the scope of this module.



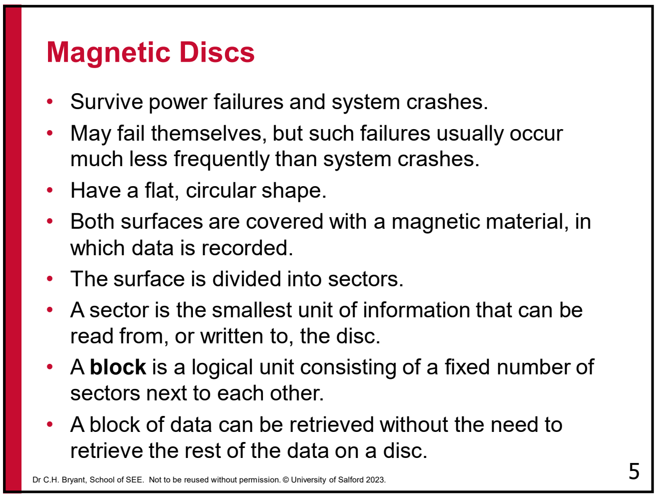
Contents

- Introduction
- Estimating costs using statistical information.
- Estimating the cost saving of a projection.
- Indexes
- Summary
- Further Reading



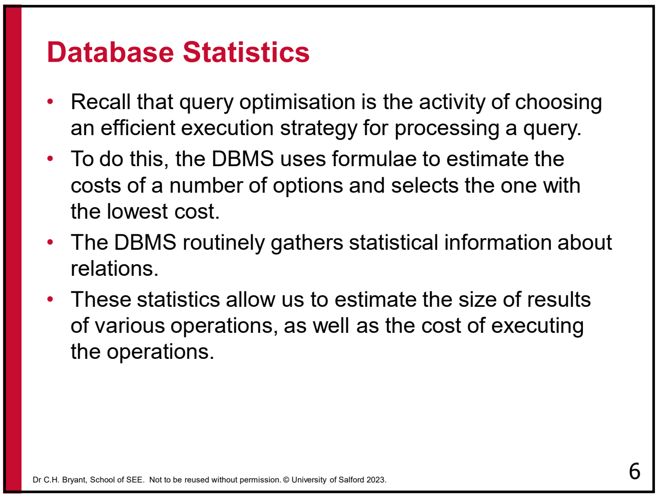
Data Storage

- **Primary Storage** - data is accessed directly by the CPU in the form of main memory (or cache memory).
  - Provides fast access.
  - Has limited capacity.
  - Is **volatile** (stored data is lost if power is cut).
- **Secondary Storage** - data is not directly accessed by the CPU so it needs to be loaded into primary memory from devices like magnetic disks.
  - Slower access than primary storage.
  - Has unlimited capacity.
  - Is **non-volatile** (it retains stored data even if power is cut).
- Usually the entire database is stored permanently on secondary storage.



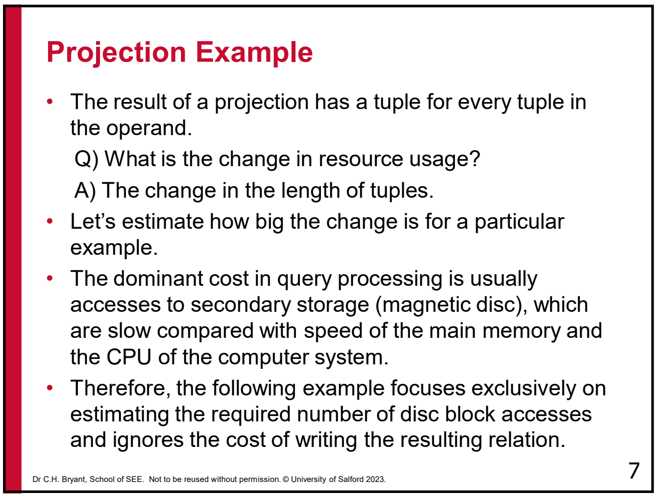
Magnetic Discs

- Survive power failures and system crashes.
- May fail themselves, but such failures usually occur much less frequently than system crashes.
- Have a flat, circular shape.
- Both surfaces are covered with a magnetic material, in which data is recorded.
- The surface is divided into sectors.
- A sector is the smallest unit of information that can be read from, or written to, the disc.
- A **block** is a logical unit consisting of a fixed number of sectors next to each other.
- A block of data can be retrieved without the need to retrieve the rest of the data on a disc.



Database Statistics

- Recall that query optimisation is the activity of choosing an efficient execution strategy for processing a query.
- To do this, the DBMS uses formulae to estimate the costs of a number of options and selects the one with the lowest cost.
- The DBMS routinely gathers statistical information about relations.
- These statistics allow us to estimate the size of results of various operations, as well as the cost of executing the operations.



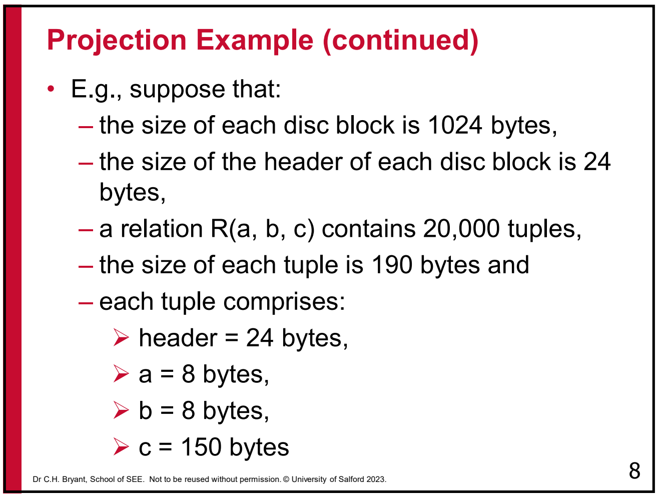
Projection Example

- The result of a projection has a tuple for every tuple in the operand.

  Q) What is the change in resource usage?

  A) The change in the length of tuples.
- Let’s estimate how big the change is for a particular example.
- The dominant cost in query processing is usually accesses to secondary storage (magnetic disc), which are slow compared with speed of the main memory and the CPU of the computer system.
- Therefore, the following example focuses exclusively on estimating the required number of disc block accesses and ignores the cost of writing the resulting relation.

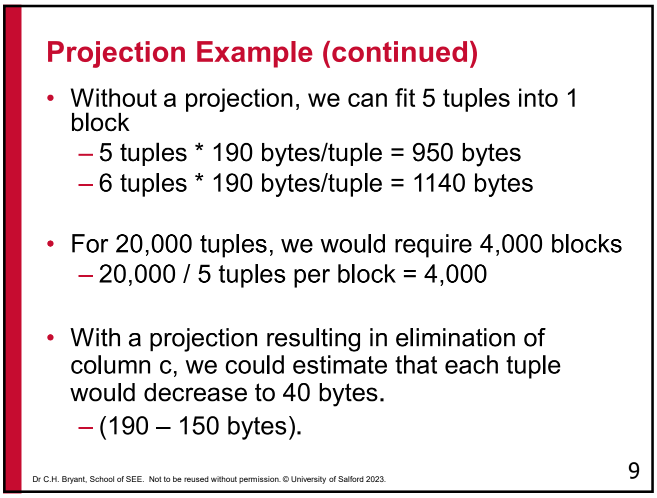
7



Projection Example (continued)

- E.g., suppose that:
  - the size of each disc block is 1024 bytes,
  - the size of the header of each disc block is 24 bytes,
  - a relation R(a, b, c) contains 20,000 tuples,
  - the size of each tuple is 190 bytes and
  - each tuple comprises:
    - header = 24 bytes,
    - a = 8 bytes,
    - b = 8 bytes,
    - c = 150 bytes

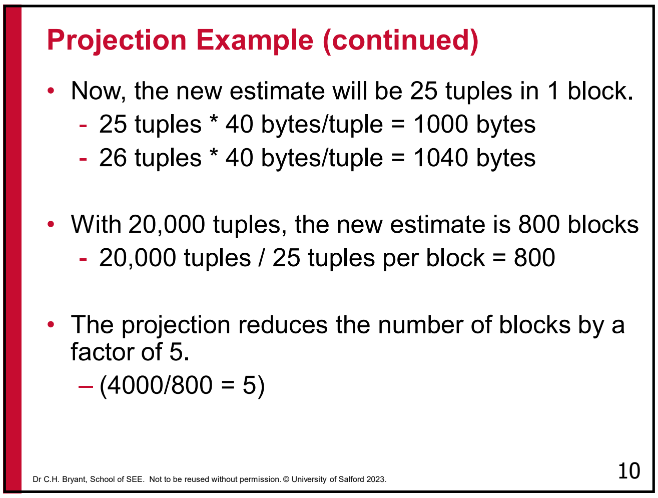
8



Projection Example (continued)

- Without a projection, we can fit 5 tuples into 1 block
  - 5 tuples * 190 bytes/tuple = 950 bytes
  - 6 tuples * 190 bytes/tuple = 1140 bytes
- For 20,000 tuples, we would require 4,000 blocks
  - 20,000 / 5 tuples per block = 4,000
- With a projection resulting in elimination of column c, we could estimate that each tuple would decrease to 40 bytes.
  - (190 – 150 bytes).

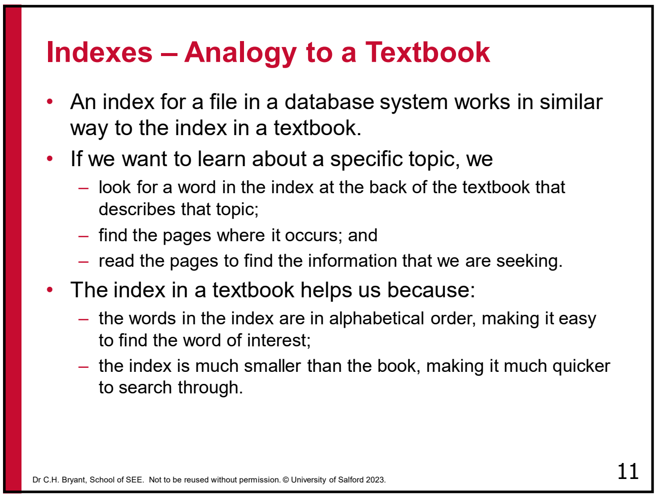
9



Projection Example (continued)

- Now, the new estimate will be 25 tuples in 1 block.
  - 25 tuples * 40 bytes/tuple = 1000 bytes
  - 26 tuples * 40 bytes/tuple = 1040 bytes
- With 20,000 tuples, the new estimate is 800 blocks
  - 20,000 tuples / 25 tuples per block = 800
- The projection reduces the number of blocks by a factor of 5.
  - (4000/800 = 5)

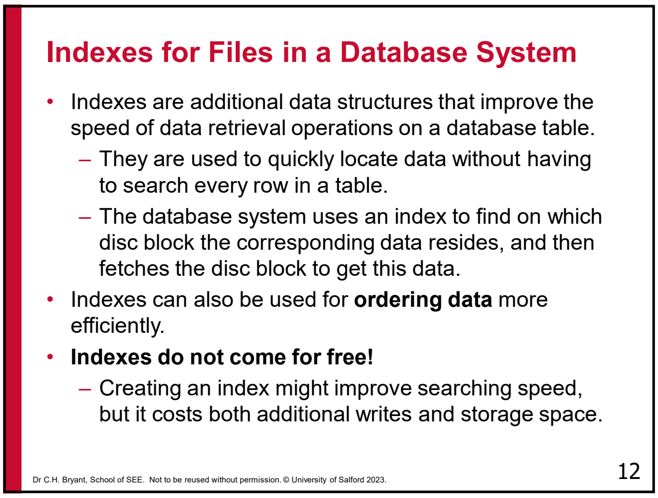
10



Indexes – Analogy to a Textbook

- An index for a file in a database system works in similar way to the index in a textbook.
- If we want to learn about a specific topic, we
  - look for a word in the index at the back of the textbook that describes that topic;
  - find the pages where it occurs; and
  - read the pages to find the information that we are seeking.
- The index in a textbook helps us because:
  - the words in the index are in alphabetical order, making it easy to find the word of interest;
  - the index is much smaller than the book, making it much quicker to search through.

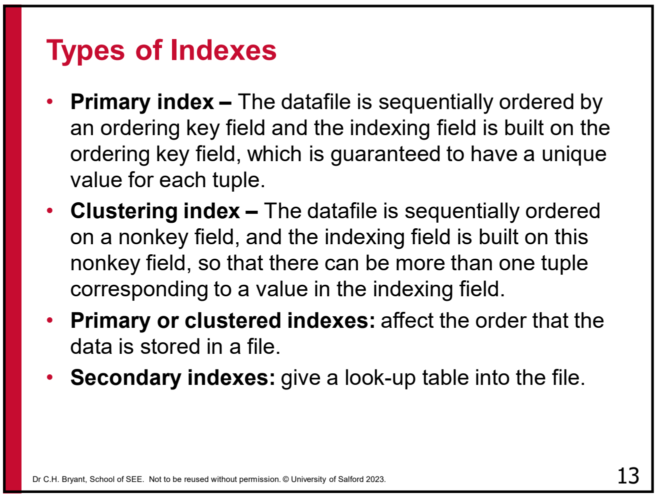
11



Indexes for Files in a Database System

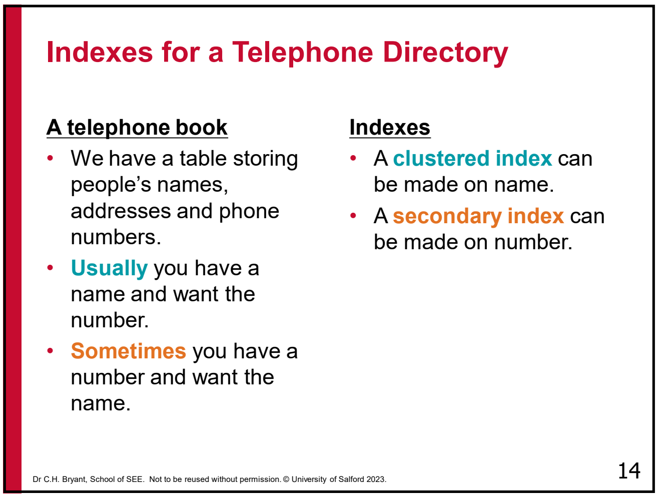
- Indexes are additional data structures that improve the speed of data retrieval operations on a database table.
  - They are used to quickly locate data without having to search every row in a table.
  - The database system uses an index to find on which disc block the corresponding data resides, and then fetches the disc block to get this data.
- Indexes can also be used for **ordering data** more efficiently.
- **Indexes do not come for free!**
  - Creating an index might improve searching speed, but it costs both additional writes and storage space.

12



Types of Indexes

- **Primary index –** The datafile is sequentially ordered by an ordering key field and the indexing field is built on the ordering key field, which is guaranteed to have a unique value for each tuple.
- **Clustering index –** The datafile is sequentially ordered on a nonkey field, and the indexing field is built on this nonkey field, so that there can be more than one tuple corresponding to a value in the indexing field.
- **Primary or clustered indexes:** affect the order that the data is stored in a file.
- **Secondary indexes:** give a look-up table into the file.



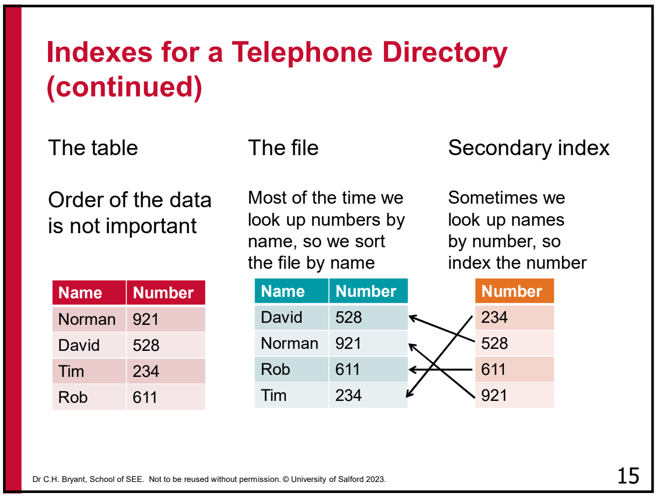
Indexes for a Telephone Directory

A telephone book

- We have a table storing people's names, addresses and phone numbers.
- **Usually** you have a name and want the number.
- **Sometimes** you have a number and want the name.

Indexes

- A **clustered index** can be made on name.
- A **secondary index** can be made on number.



Indexes for a Telephone Directory (continued)

The table

Order of the data is not important

Name	Number
Norman	921
David	528
Tim	234
Rob	611

The file

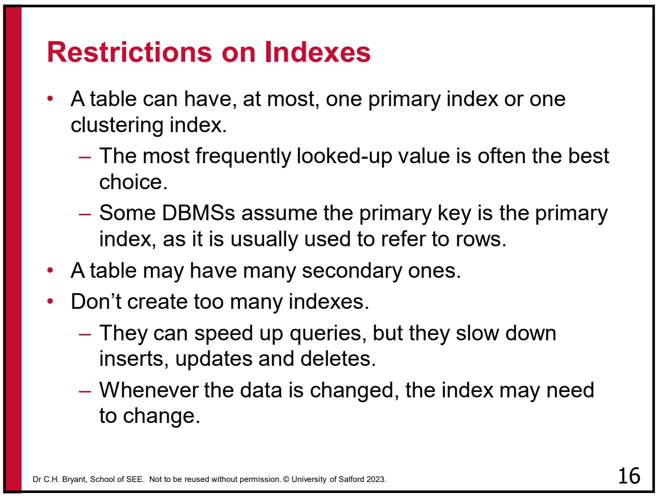
Most of the time we look up numbers by name, so we sort the file by name

Name	Number
David	528
Norman	921
Rob	611
Tim	234

Secondary index

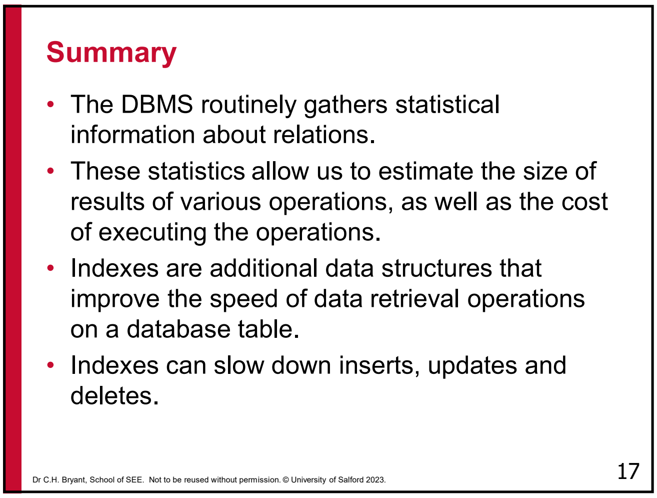
Sometimes we look up names by number, so index the number

Number
234
528
611
921



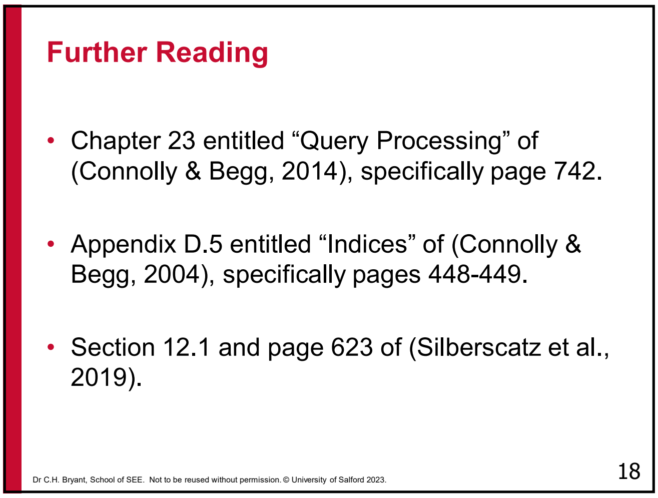
Restrictions on Indexes

- A table can have, at most, one primary index or one clustering index.
  - The most frequently looked-up value is often the best choice.
  - Some DBMSs assume the primary key is the primary index, as it is usually used to refer to rows.
- A table may have many secondary ones.
- Don't create too many indexes.
  - They can speed up queries, but they slow down inserts, updates and deletes.
  - Whenever the data is changed, the index may need to change.



Summary

- The DBMS routinely gathers statistical information about relations.
- These statistics allow us to estimate the size of results of various operations, as well as the cost of executing the operations.
- Indexes are additional data structures that improve the speed of data retrieval operations on a database table.
- Indexes can slow down inserts, updates and deletes.



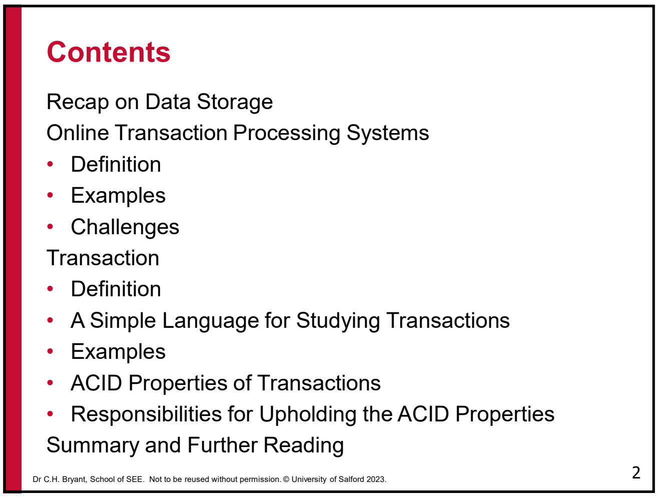
Further Reading

- Chapter 23 entitled "Query Processing" of (Connolly & Begg, 2014), specifically page 742.
- Appendix D.5 entitled "Indices" of (Connolly & Begg, 2004), specifically pages 448-449.
- Section 12.1 and page 623 of (Silberscatz et al., 2019).

University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: TRANSACTIONS



Contents

Recap on Data Storage

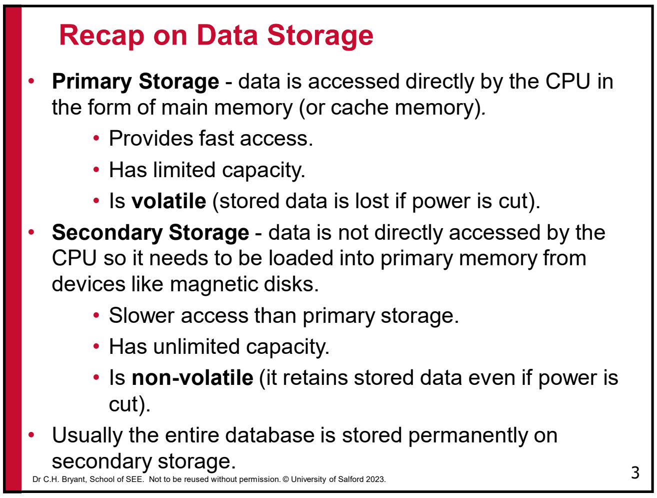
Online Transaction Processing Systems

- Definition
- Examples
- Challenges

Transaction

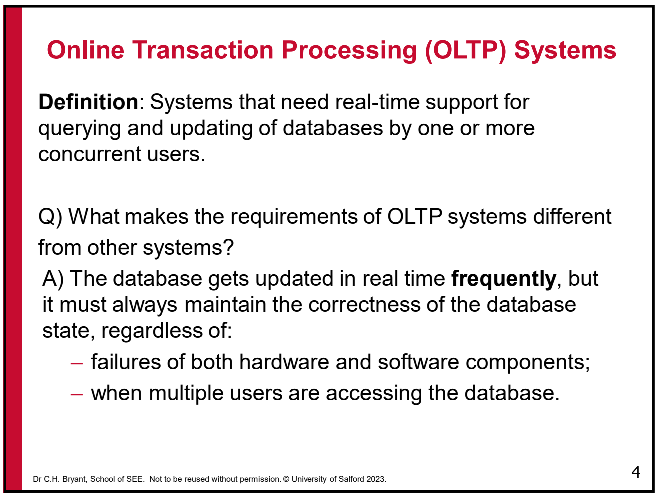
- Definition
- A Simple Language for Studying Transactions
- Examples
- ACID Properties of Transactions
- Responsibilities for Upholding the ACID Properties

Summary and Further Reading



Recap on Data Storage

- **Primary Storage** - data is accessed directly by the CPU in the form of main memory (or cache memory).
  - Provides fast access.
  - Has limited capacity.
  - Is **volatile** (stored data is lost if power is cut).
- **Secondary Storage** - data is not directly accessed by the CPU so it needs to be loaded into primary memory from devices like magnetic disks.
  - Slower access than primary storage.
  - Has unlimited capacity.
  - Is **non-volatile** (it retains stored data even if power is cut).
- Usually the entire database is stored permanently on secondary storage.



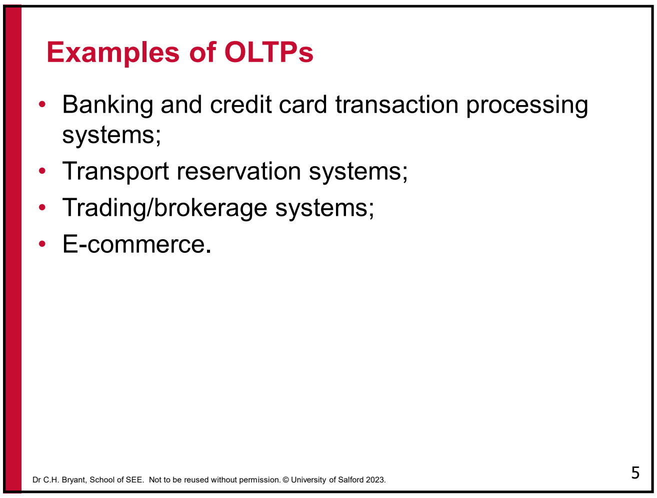
Online Transaction Processing (OLTP) Systems

Definition: Systems that need real-time support for querying and updating of databases by one or more concurrent users.

Q) What makes the requirements of OLTP systems different from other systems?

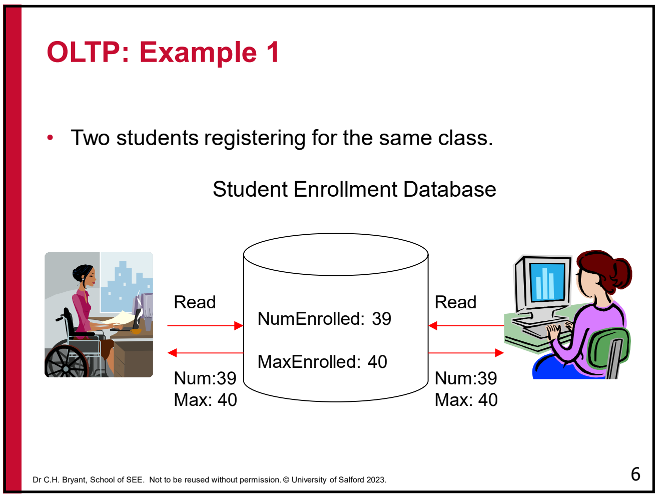
A) The database gets updated in real time **frequently**, but it must always maintain the correctness of the database state, regardless of:

- failures of both hardware and software components;
- when multiple users are accessing the database.



Examples of OLTPs

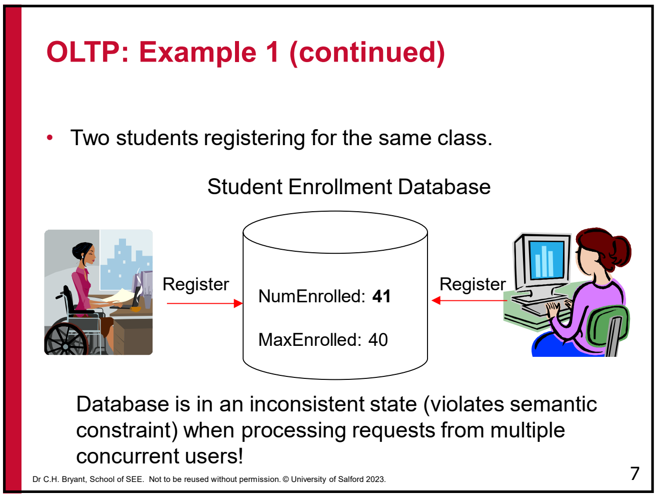
- Banking and credit card transaction processing systems;
- Transport reservation systems;
- Trading/brokerage systems;
- E-commerce.



OLTP: Example 1

- Two students registering for the same class.

Student Enrollment Database

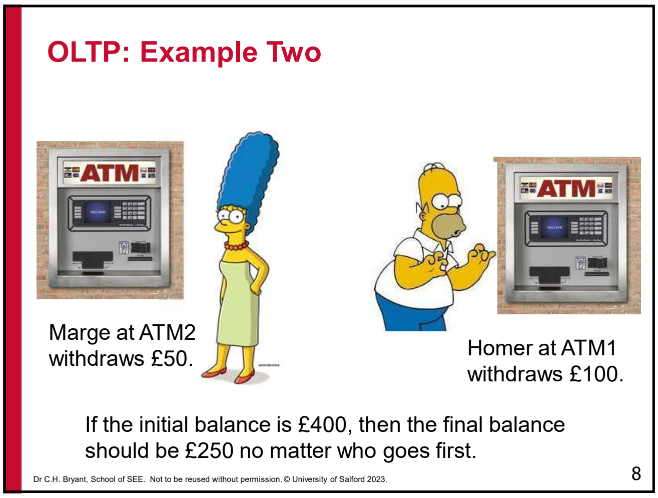


OLTP: Example 1 (continued)

- Two students registering for the same class.

Database is in an inconsistent state (violates semantic constraint) when processing requests from multiple concurrent users!

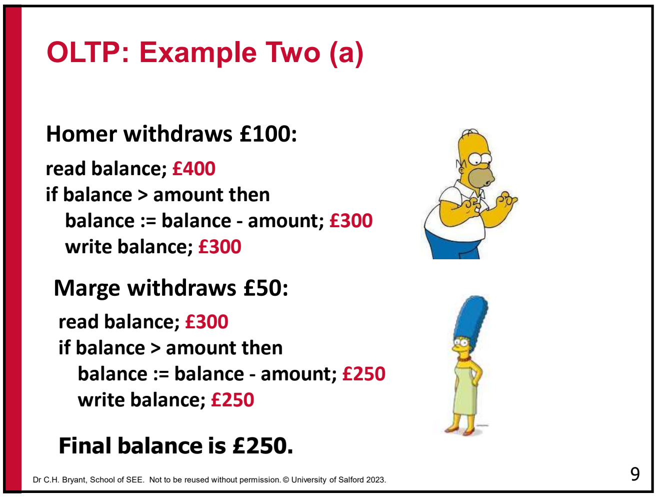
7



OLTP: Example Two

If the initial balance is £400, then the final balance should be £250 no matter who goes first.

8



OLTP: Example Two (a)

Homer withdraws £100:

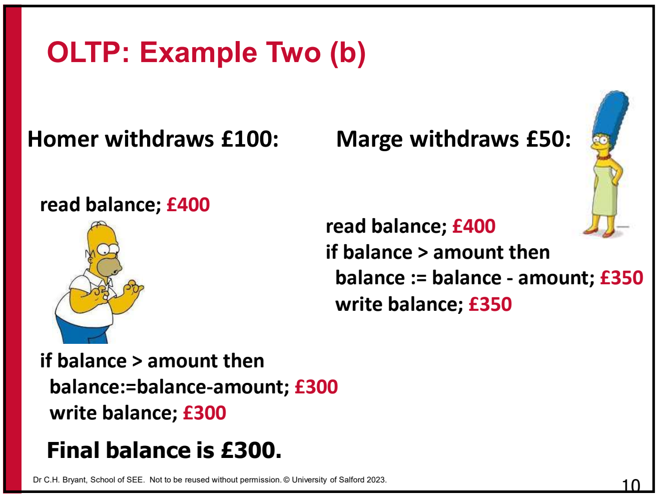
```
read balance; £400
if balance > amount then
   balance := balance - amount; £300
   write balance; £300
```

Marge withdraws £50:

```
read balance; £300
if balance > amount then
   balance := balance - amount; £250
   write balance; £250
```

Final balance is £250.

9



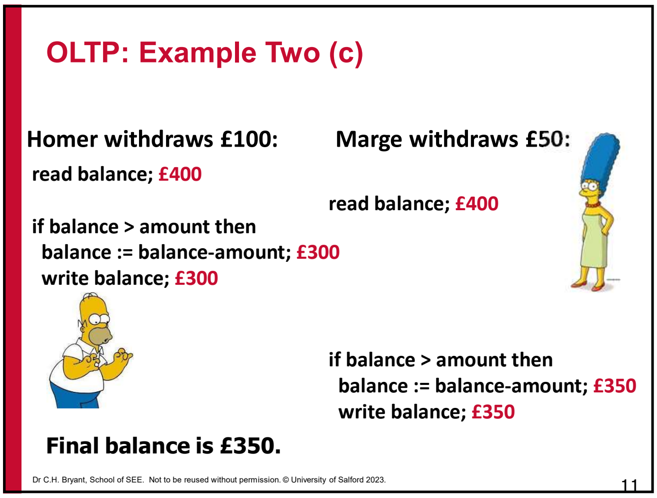
OLTP: Example Two (b)

Homer withdraws £100: **Marge withdraws £50:**

```
read balance; £400
                              read balance; £400
                              if balance > amount then
                                 balance := balance - amount; £350
                                 write balance; £350
if balance > amount then
  balance:=balance-amount; £300
  write balance; £300
```

Final balance is £300.

10



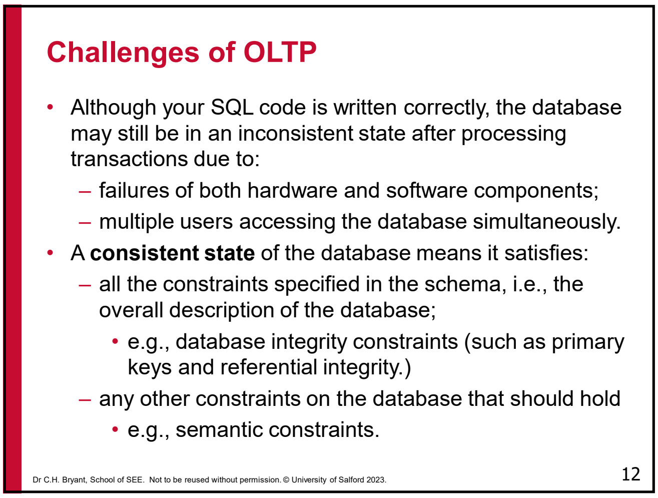
OLTP: Example Two (c)

Homer withdraws £100: **Marge withdraws £50:**

```
read balance; £400
                              read balance; £400
if balance > amount then
  balance := balance-amount; £300
  write balance; £300
                              if balance > amount then
                                balance := balance-amount; £350
                                write balance; £350
```

Final balance is £350.

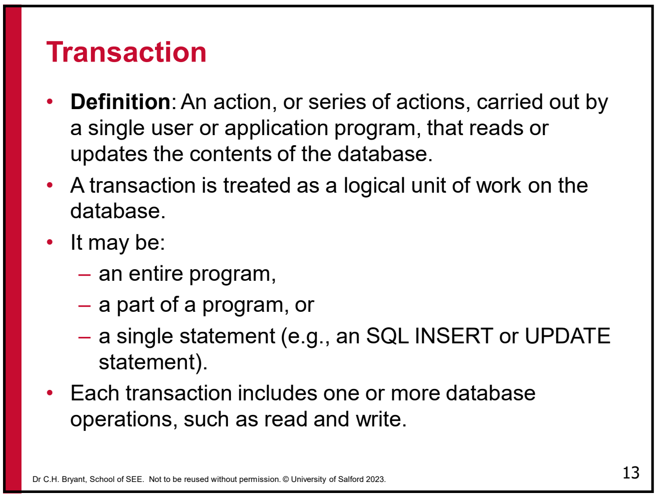
11



Challenges of OLTP

- Although your SQL code is written correctly, the database may still be in an inconsistent state after processing transactions due to:
  - failures of both hardware and software components;
  - multiple users accessing the database simultaneously.
- A **consistent state** of the database means it satisfies:
  - all the constraints specified in the schema, i.e., the overall description of the database;
    - e.g., database integrity constraints (such as primary keys and referential integrity.)
  - any other constraints on the database that should hold
    - e.g., semantic constraints.

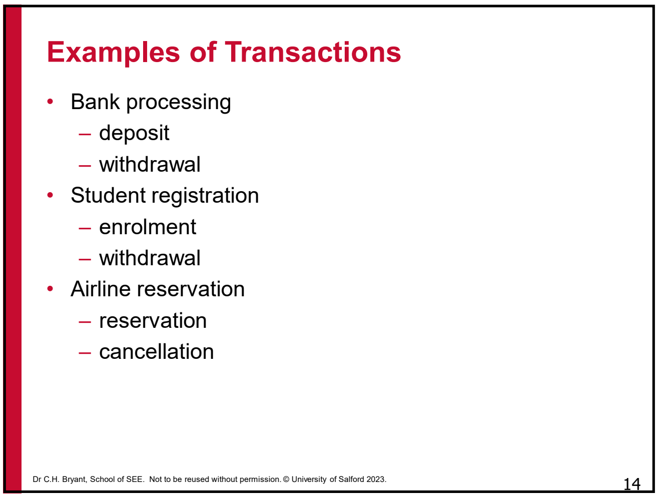
12



Transaction

- **Definition**: An action, or series of actions, carried out by a single user or application program, that reads or updates the contents of the database.
- A transaction is treated as a logical unit of work on the database.
- It may be:
  - an entire program,
  - a part of a program, or
  - a single statement (e.g., an SQL INSERT or UPDATE statement).
- Each transaction includes one or more database operations, such as read and write.

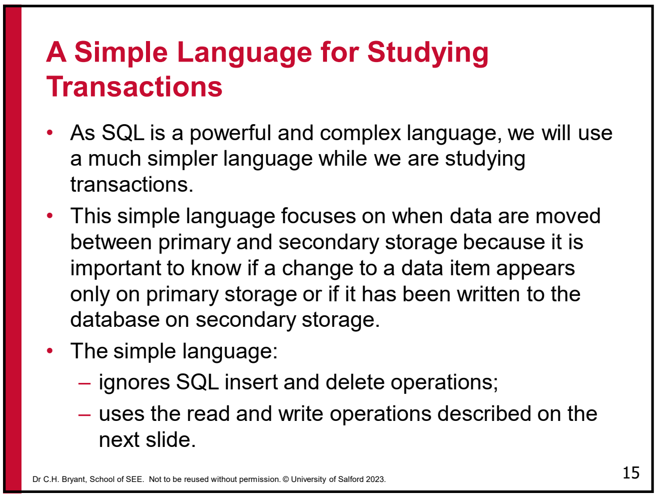
13



Examples of Transactions

- Bank processing
  - deposit
  - withdrawal
- Student registration
  - enrolment
  - withdrawal
- Airline reservation
  - reservation
  - cancellation

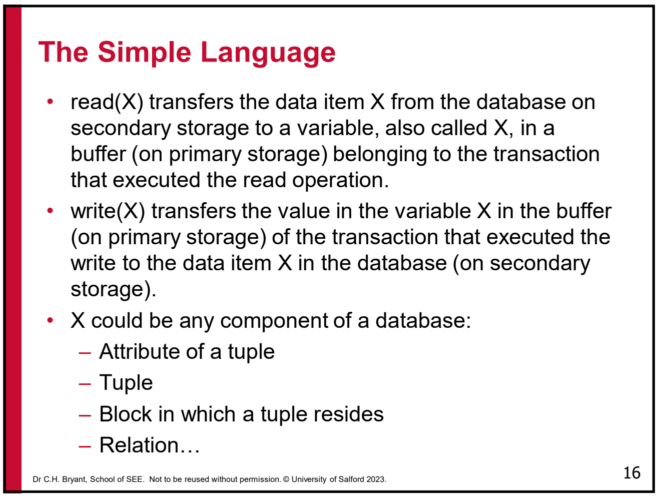
14



A Simple Language for Studying Transactions

- As SQL is a powerful and complex language, we will use a much simpler language while we are studying transactions.
- This simple language focuses on when data are moved between primary and secondary storage because it is important to know if a change to a data item appears only on primary storage or if it has been written to the database on secondary storage.
- The simple language:
  - ignores SQL insert and delete operations;
  - uses the read and write operations described on the next slide.

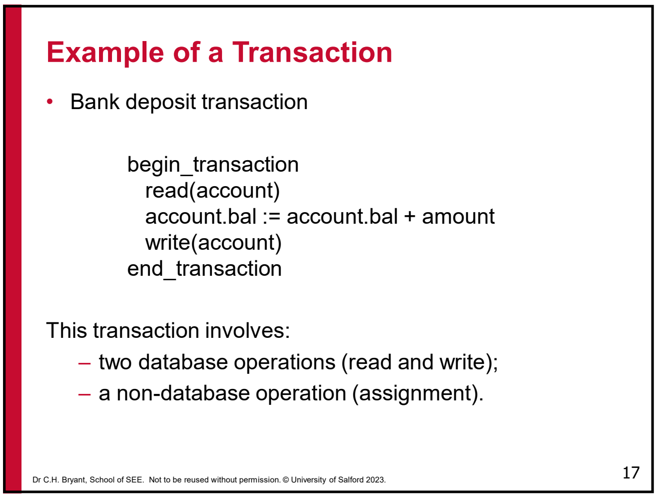
15



The Simple Language

- read(X) transfers the data item X from the database on secondary storage to a variable, also called X, in a buffer (on primary storage) belonging to the transaction that executed the read operation.
- write(X) transfers the value in the variable X in the buffer (on primary storage) of the transaction that executed the write to the data item X in the database (on secondary storage).
- X could be any component of a database:
  - Attribute of a tuple
  - Tuple
  - Block in which a tuple resides
  - Relation…

16



Example of a Transaction

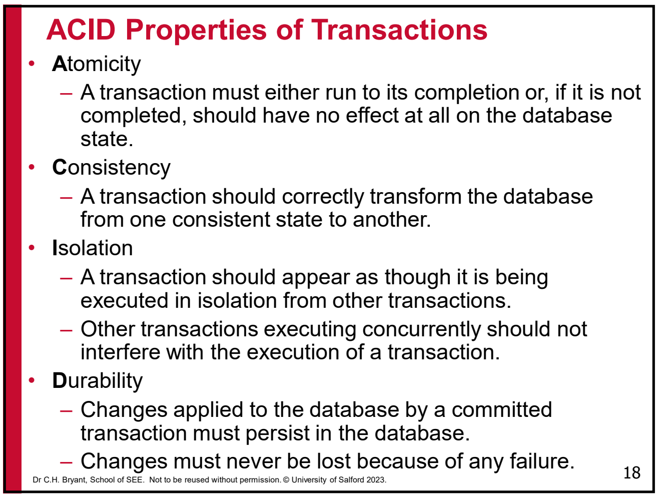
- Bank deposit transaction

```
begin_transaction
  read(account)
  account.bal := account.bal + amount
  write(account)
end_transaction
```

This transaction involves:

- two database operations (read and write);
- a non-database operation (assignment).

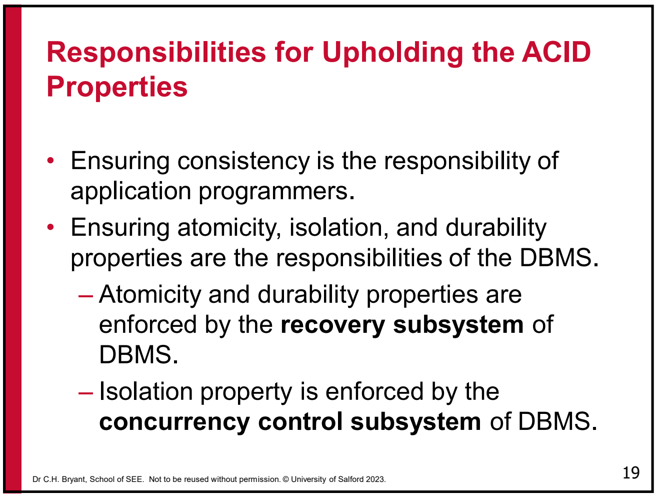
17



ACID Properties of Transactions

- **A**tomicity
  - A transaction must either run to its completion or, if it is not completed, should have no effect at all on the database state.
- **C**onsistency
  - A transaction should correctly transform the database from one consistent state to another.
- **I**solation
  - A transaction should appear as though it is being executed in isolation from other transactions.
  - Other transactions executing concurrently should not interfere with the execution of a transaction.
- **D**urability
  - Changes applied to the database by a committed transaction must persist in the database.
  - Changes must never be lost because of any failure.

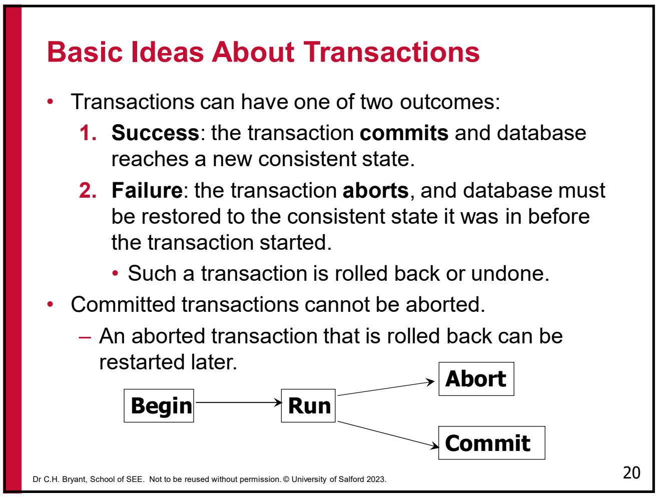
18



Responsibilities for Upholding the ACID Properties

- Ensuring consistency is the responsibility of application programmers.
- Ensuring atomicity, isolation, and durability properties are the responsibilities of the DBMS.
  - Atomicity and durability properties are enforced by the **recovery subsystem** of DBMS.
  - Isolation property is enforced by the **concurrency control subsystem** of DBMS.

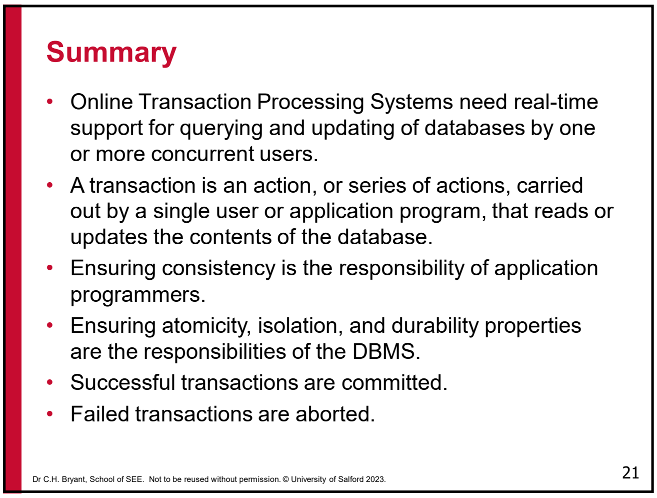
19



Basic Ideas About Transactions

- Transactions can have one of two outcomes:
  1. **Success**: the transaction **commits** and database reaches a new consistent state.
  2. **Failure**: the transaction **aborts**, and database must be restored to the consistent state it was in before the transaction started.
     - Such a transaction is rolled back or undone.
- Committed transactions cannot be aborted.
  - An aborted transaction that is rolled back can be restarted later.

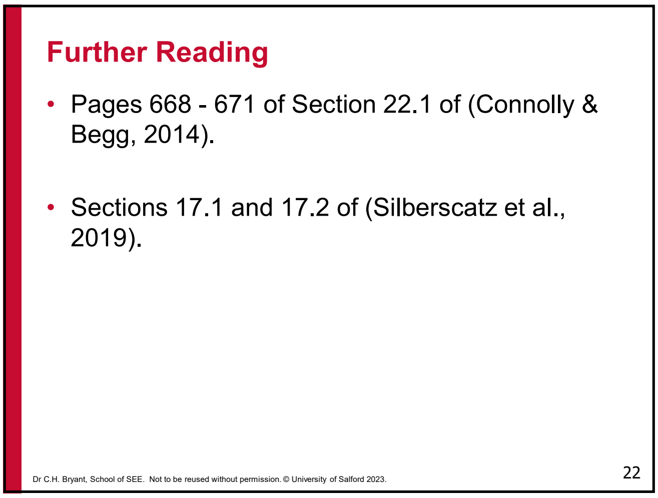
20



Summary

- Online Transaction Processing Systems need real-time support for querying and updating of databases by one or more concurrent users.
- A transaction is an action, or series of actions, carried out by a single user or application program, that reads or updates the contents of the database.
- Ensuring consistency is the responsibility of application programmers.
- Ensuring atomicity, isolation, and durability properties are the responsibilities of the DBMS.
- Successful transactions are committed.
- Failed transactions are aborted.

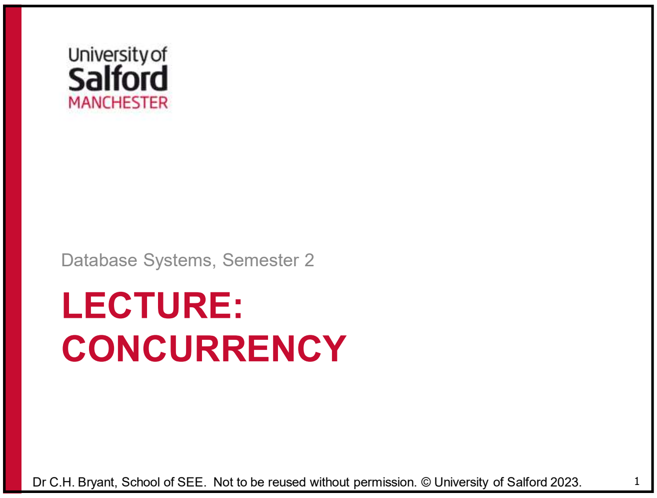
21



Further Reading

- Pages 668 - 671 of Section 22.1 of (Connolly & Begg, 2014).
- Sections 17.1 and 17.2 of (Silberscatz et al., 2019).

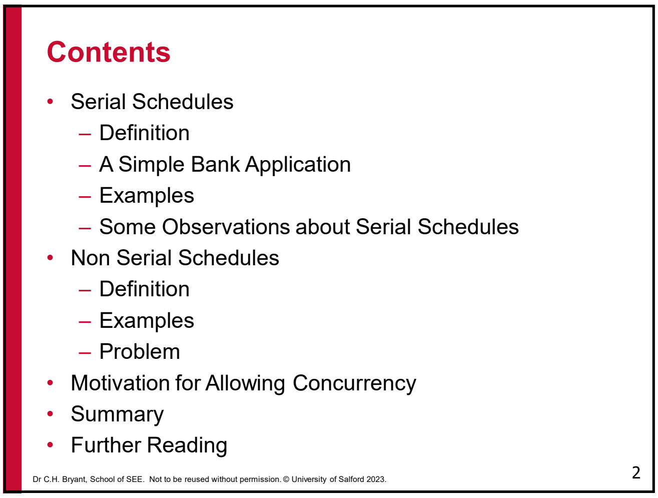
22



University of Salford MANCHESTER

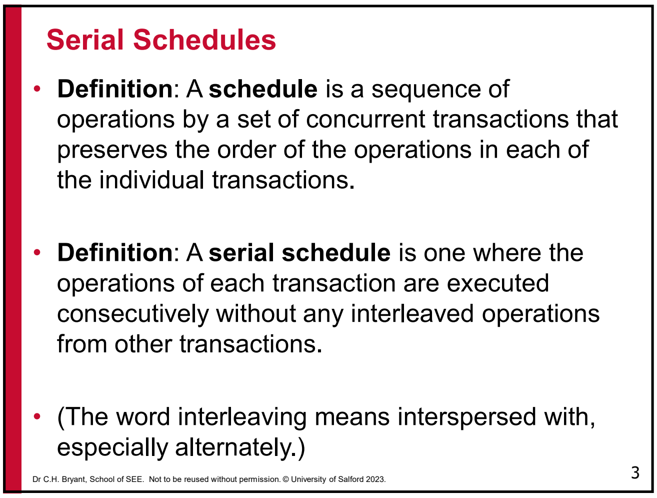
Database Systems, Semester 2

LECTURE: CONCURRENCY



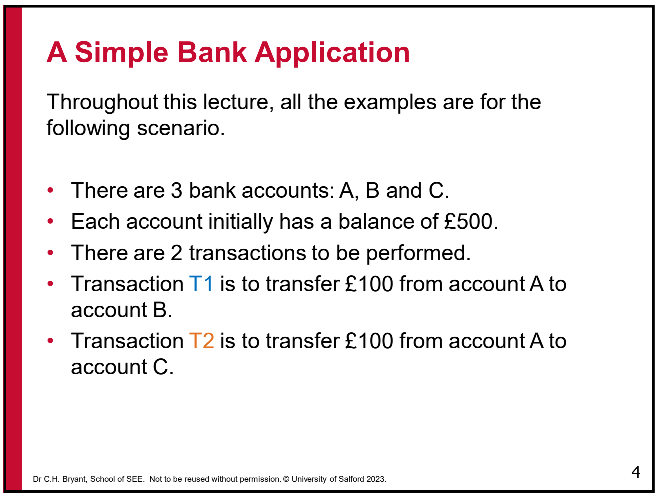
Contents

- Serial Schedules
  - Definition
  - A Simple Bank Application
  - Examples
  - Some Observations about Serial Schedules
- Non Serial Schedules
  - Definition
  - Examples
  - Problem
- Motivation for Allowing Concurrency
- Summary
- Further Reading



Serial Schedules

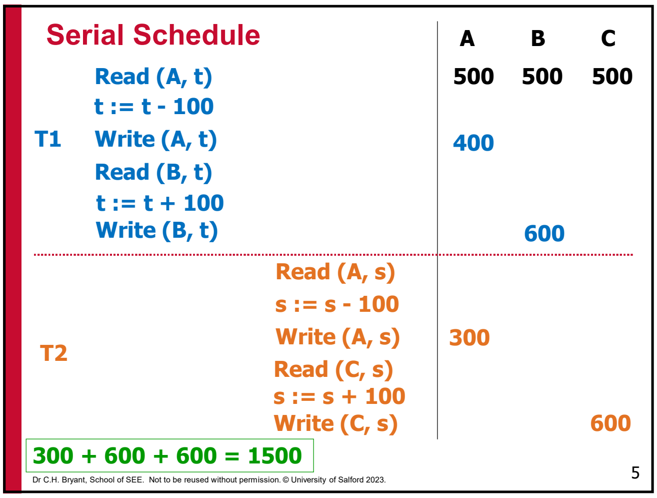
- **Definition**: A **schedule** is a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- **Definition**: A **serial schedule** is one where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- (The word interleaving means interspersed with, especially alternately.)



A Simple Bank Application

Throughout this lecture, all the examples are for the following scenario.

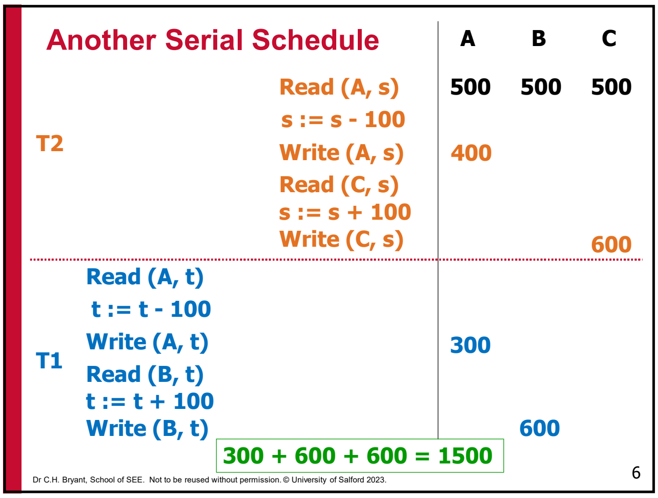
- There are 3 bank accounts: A, B and C.
- Each account initially has a balance of £500.
- There are 2 transactions to be performed.
- Transaction T1 is to transfer £100 from account A to account B.
- Transaction T2 is to transfer £100 from account A to account C.



Serial Schedule

			A	B	C
	Read (A, t)		500	500	500
	t := t - 100				
T1	Write (A, t)		400		
	Read (B, t)				
	t := t + 100				
	Write (B, t)			600	
		Read (A, s)			
		s := s - 100			
		Write (A, s)	300		
T2		Read (C, s)			
		s := s + 100			
		Write (C, s)			600

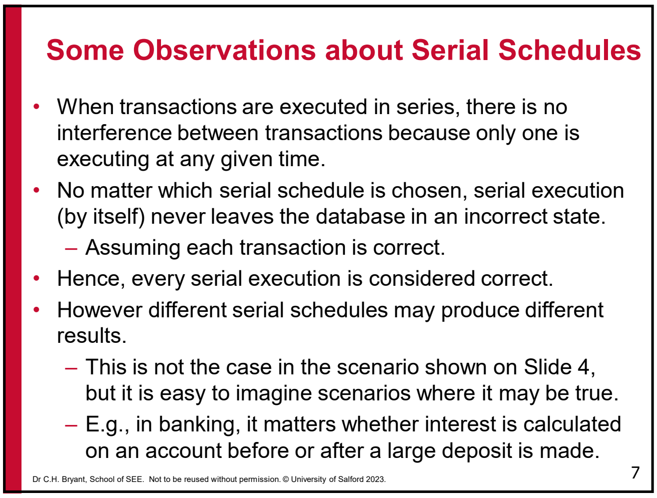
300 + 600 + 600 = 1500



Another Serial Schedule

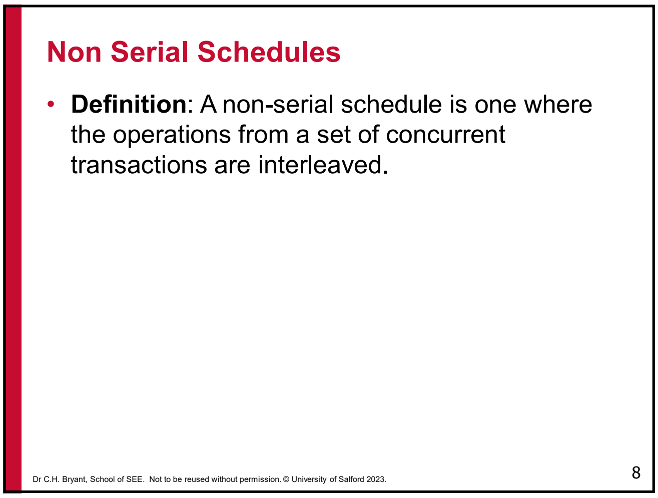
			A	B	C
		Read (A, s)	500	500	500
T2		s := s - 100			
		Write (A, s)	400		
		Read (C, s)			
		s := s + 100			
		Write (C, s)			600
	Read (A, t)				
	t := t - 100				
	Write (A, t)		300		
T1	Read (B, t)				
	t := t + 100				
	Write (B, t)			600	

300 + 600 + 600 = 1500



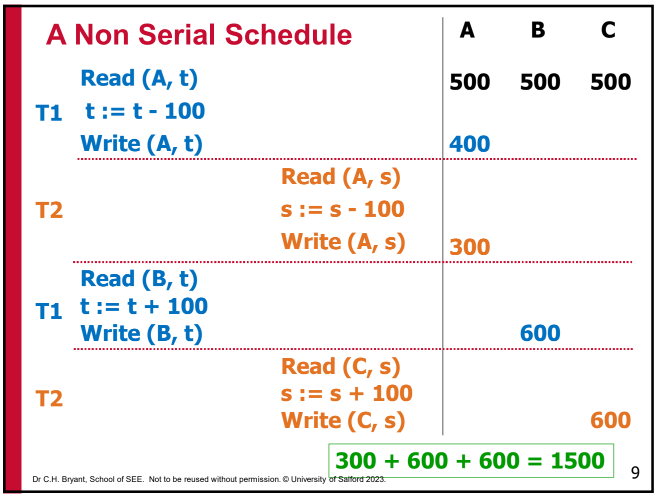
Some Observations about Serial Schedules

- When transactions are executed in series, there is no interference between transactions because only one is executing at any given time.
- No matter which serial schedule is chosen, serial execution (by itself) never leaves the database in an incorrect state.
  - Assuming each transaction is correct.
- Hence, every serial execution is considered correct.
- However different serial schedules may produce different results.
  - This is not the case in the scenario shown on Slide 4, but it is easy to imagine scenarios where it may be true.
  - E.g., in banking, it matters whether interest is calculated on an account before or after a large deposit is made.



Non Serial Schedules

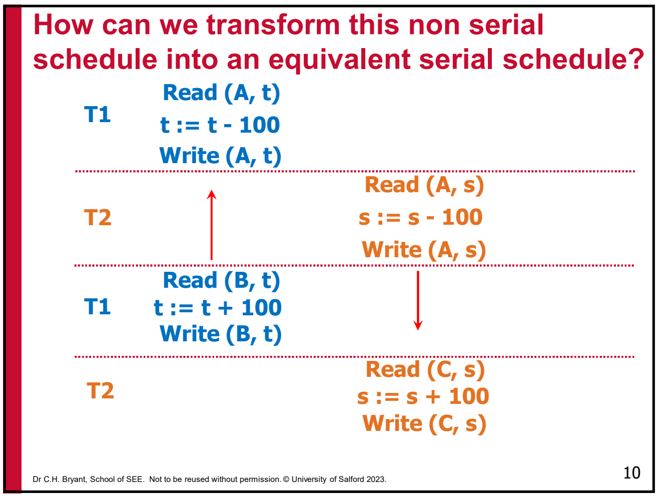
- **Definition**: A non-serial schedule is one where the operations from a set of concurrent transactions are interleaved.



A Non Serial Schedule

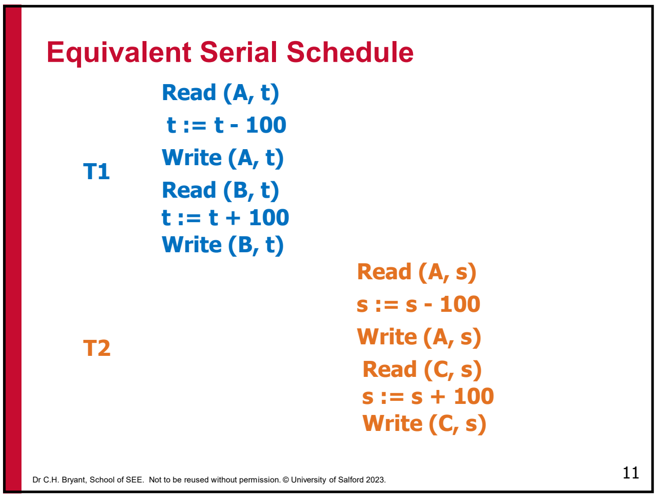
			A	B	C
			500	500	500
T1	Read (A, t) t := t - 100 Write (A, t)		400		
T2		Read (A, s) s := s - 100 Write (A, s)	300		
T1	Read (B, t) t := t + 100 Write (B, t)			600	
T2		Read (C, s) s := s + 100 Write (C, s)			600

300 + 600 + 600 = 1500



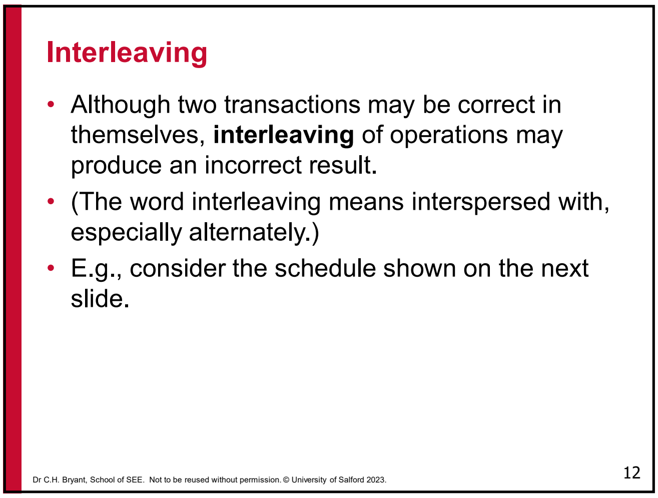
How can we transform this non serial schedule into an equivalent serial schedule?

T1	Read (A, t) t := t - 100 Write (A, t)	
T2	↑	Read (A, s) s := s - 100 Write (A, s)
T1	Read (B, t) t := t + 100 Write (B, t)	↓
T2		Read (C, s) s := s + 100 Write (C, s)



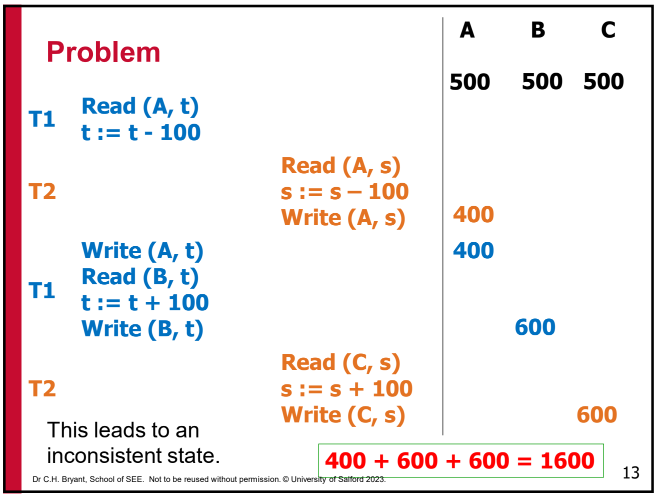
Equivalent Serial Schedule

T1	Read (A, t) t := t - 100 Write (A, t) Read (B, t) t := t + 100 Write (B, t)	
T2		Read (A, s) s := s - 100 Write (A, s) Read (C, s) s := s + 100 Write (C, s)



Interleaving

- Although two transactions may be correct in themselves, **interleaving** of operations may produce an incorrect result.
- (The word interleaving means interspersed with, especially alternately.)
- E.g., consider the schedule shown on the next slide.

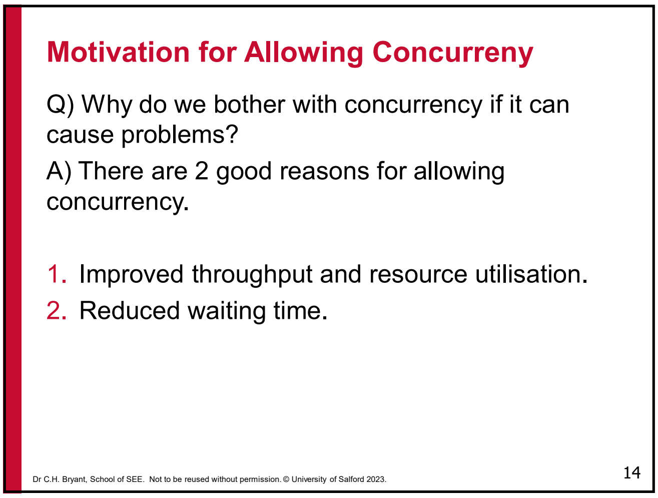


Problem

			A	B	C
			500	500	500
T1	Read (A, t) t := t - 100				
T2		Read (A, s) s := s – 100 Write (A, s)	400		
T1	Write (A, t) Read (B, t) t := t + 100 Write (B, t)		400	600	
T2		Read (C, s) s := s + 100 Write (C, s)			600

This leads to an inconsistent state.

400 + 600 + 600 = 1600

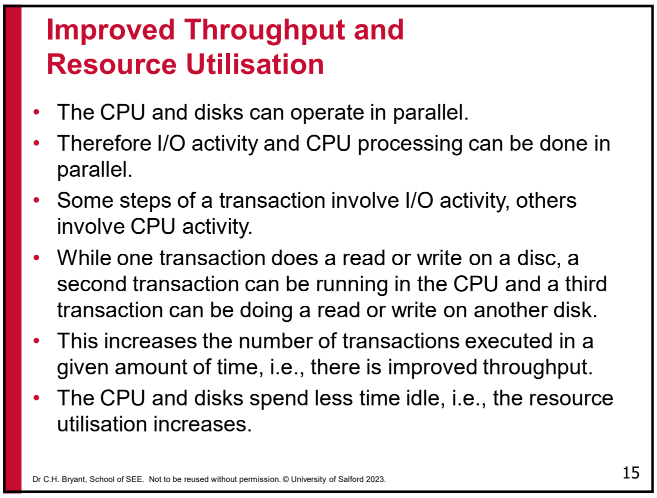


Motivation for Allowing Concurreny

Q) Why do we bother with concurrency if it can cause problems?

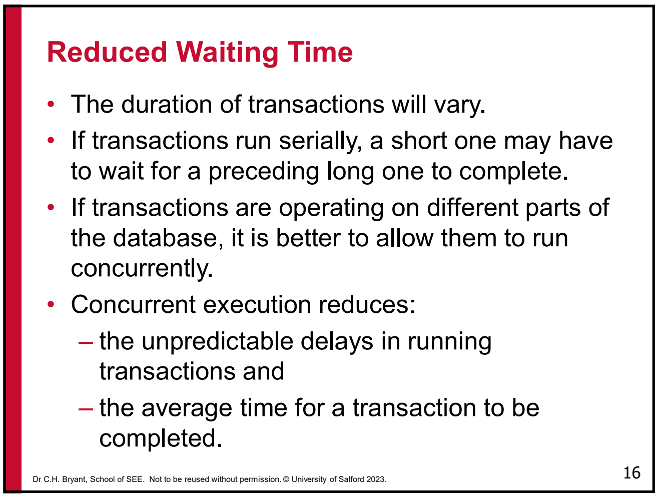
A) There are 2 good reasons for allowing concurrency.

1. Improved throughput and resource utilisation.
2. Reduced waiting time.



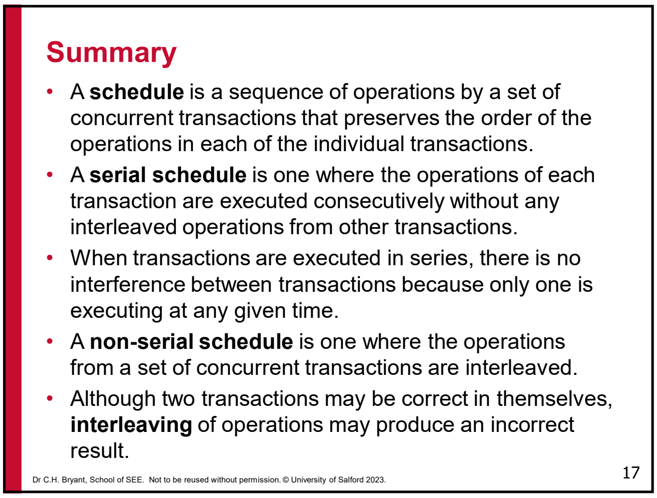
Improved Throughput and Resource Utilisation

- The CPU and disks can operate in parallel.
- Therefore I/O activity and CPU processing can be done in parallel.
- Some steps of a transaction involve I/O activity, others involve CPU activity.
- While one transaction does a read or write on a disc, a second transaction can be running in the CPU and a third transaction can be doing a read or write on another disk.
- This increases the number of transactions executed in a given amount of time, i.e., there is improved throughput.
- The CPU and disks spend less time idle, i.e., the resource utilisation increases.



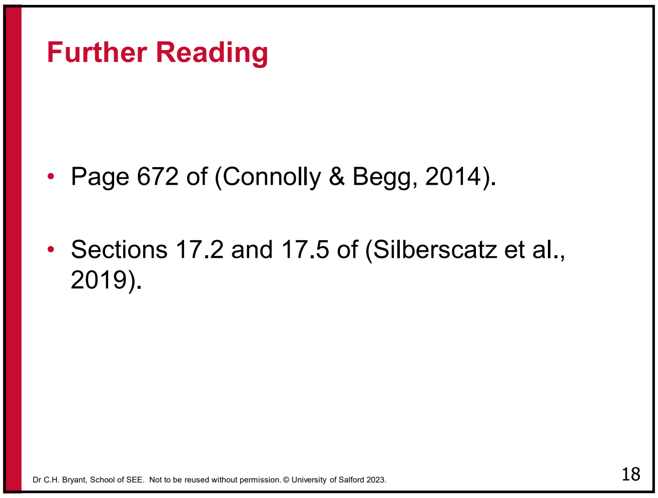
Reduced Waiting Time

- The duration of transactions will vary.
- If transactions run serially, a short one may have to wait for a preceding long one to complete.
- If transactions are operating on different parts of the database, it is better to allow them to run concurrently.
- Concurrent execution reduces:
  - the unpredictable delays in running transactions and
  - the average time for a transaction to be completed.



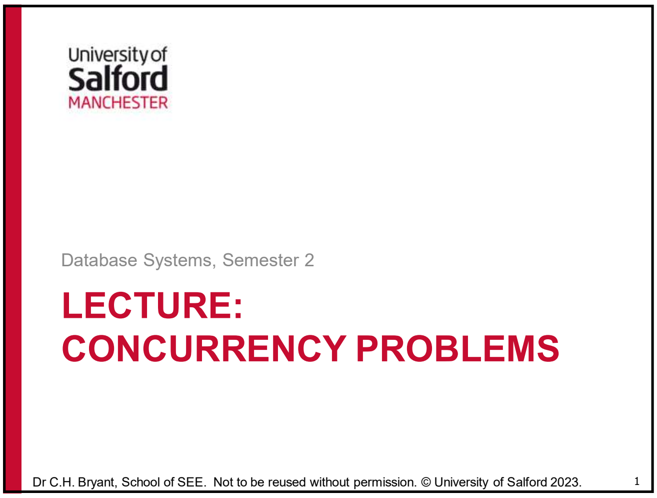
Summary

- A **schedule** is a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- A **serial schedule** is one where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- When transactions are executed in series, there is no interference between transactions because only one is executing at any given time.
- A **non-serial schedule** is one where the operations from a set of concurrent transactions are interleaved.
- Although two transactions may be correct in themselves, **interleaving** of operations may produce an incorrect result.



Further Reading

- Page 672 of (Connolly & Begg, 2014).
- Sections 17.2 and 17.5 of (Silberscatz et al., 2019).

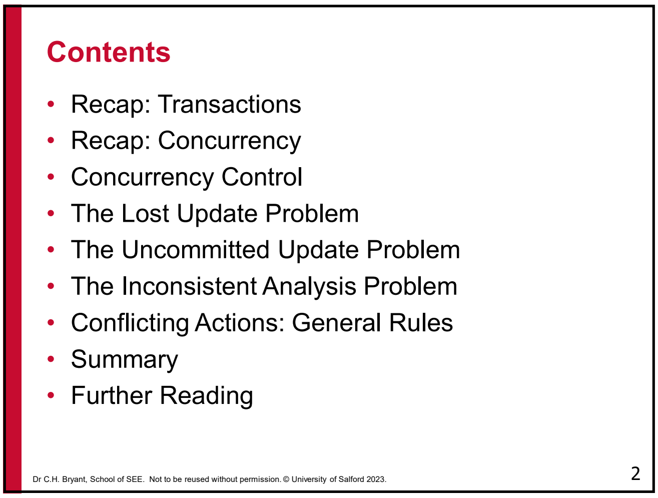


University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: CONCURRENCY PROBLEMS

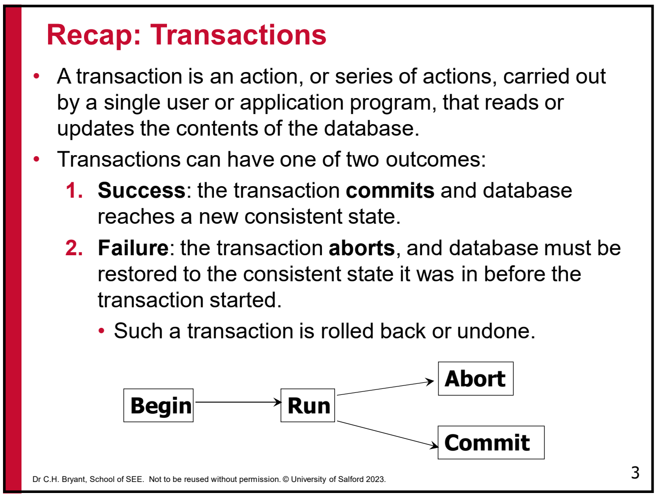
1



Contents

- Recap: Transactions
- Recap: Concurrency
- Concurrency Control
- The Lost Update Problem
- The Uncommitted Update Problem
- The Inconsistent Analysis Problem
- Conflicting Actions: General Rules
- Summary
- Further Reading

2



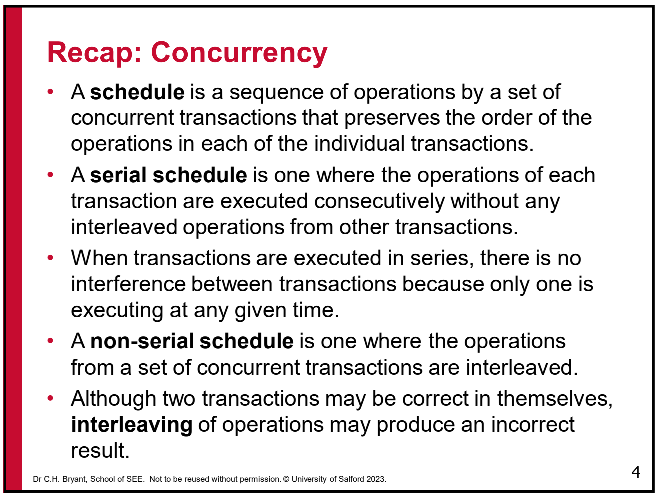
Recap: Transactions

- A transaction is an action, or series of actions, carried out by a single user or application program, that reads or updates the contents of the database.
- Transactions can have one of two outcomes:
  1. **Success**: the transaction **commits** and database reaches a new consistent state.
  2. **Failure**: the transaction **aborts**, and database must be restored to the consistent state it was in before the transaction started.
     - Such a transaction is rolled back or undone.

Begin → **Run** → **Abort**

Run → **Commit**

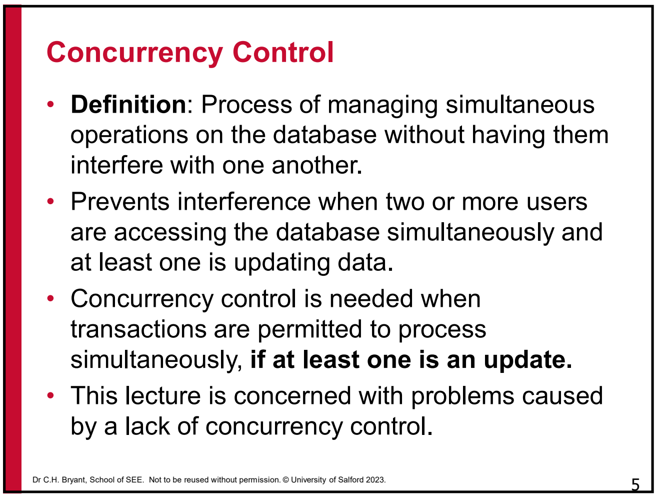
3



Recap: Concurrency

- A **schedule** is a sequence of operations by a set of concurrent transactions that preserves the order of the operations in each of the individual transactions.
- A **serial schedule** is one where the operations of each transaction are executed consecutively without any interleaved operations from other transactions.
- When transactions are executed in series, there is no interference between transactions because only one is executing at any given time.
- A **non-serial schedule** is one where the operations from a set of concurrent transactions are interleaved.
- Although two transactions may be correct in themselves, **interleaving** of operations may produce an incorrect result.

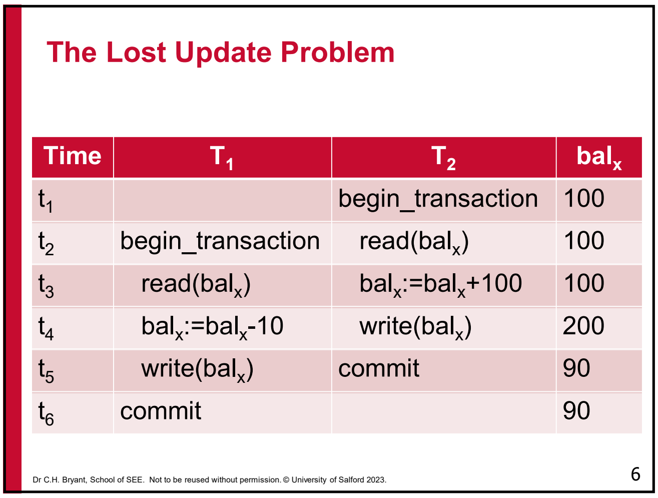
4



Concurrency Control

- **Definition**: Process of managing simultaneous operations on the database without having them interfere with one another.
- Prevents interference when two or more users are accessing the database simultaneously and at least one is updating data.
- Concurrency control is needed when transactions are permitted to process simultaneously, **if at least one is an update.**
- This lecture is concerned with problems caused by a lack of concurrency control.

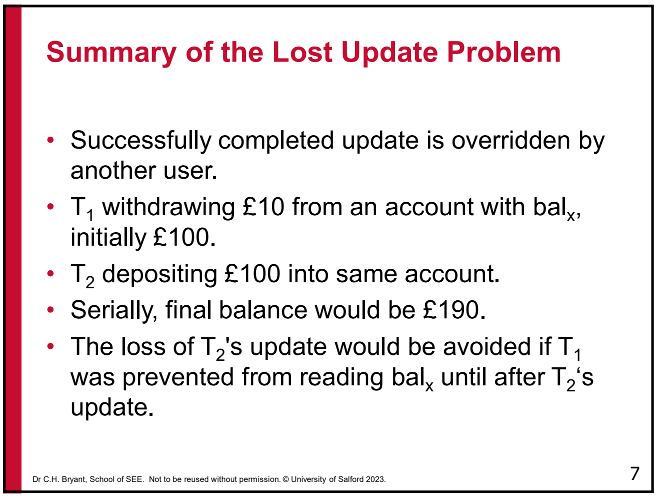
5



The Lost Update Problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	read(bal_x)	bal_x:=bal_x+100	100
t_4	bal_x:=bal_x-10	write(bal_x)	200
t_5	write(bal_x)	commit	90
t_6	commit		90

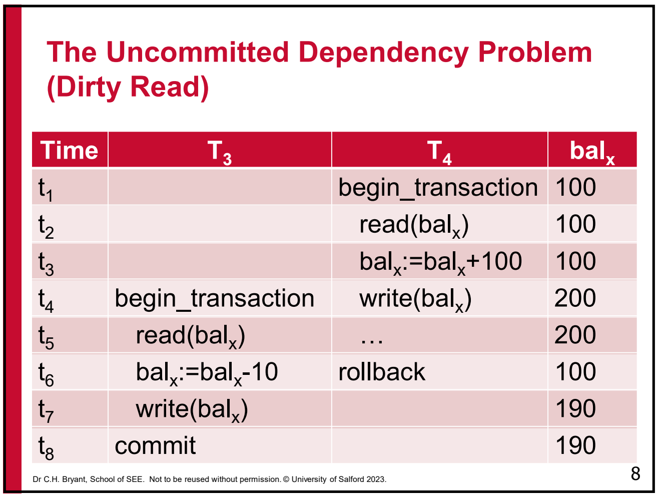
6



Summary of the Lost Update Problem

- Successfully completed update is overridden by another user.
- T_1 withdrawing £10 from an account with bal_x, initially £100.
- T_2 depositing £100 into same account.
- Serially, final balance would be £190.
- The loss of T_2's update would be avoided if T_1 was prevented from reading bal_x until after T_2‘s update.

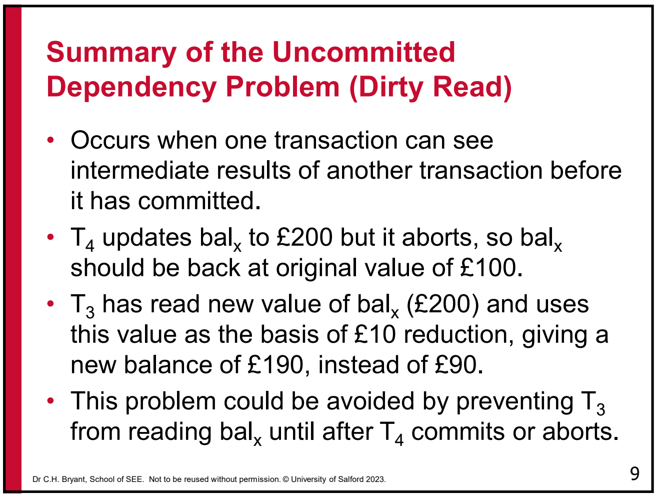
7



The Uncommitted Dependency Problem (Dirty Read)

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		bal_x:=bal_x+100	100
t_4	begin_transaction	write(bal_x)	200
t_5	read(bal_x)	…	200
t_6	bal_x:=bal_x-10	rollback	100
t_7	write(bal_x)		190
t_8	commit		190

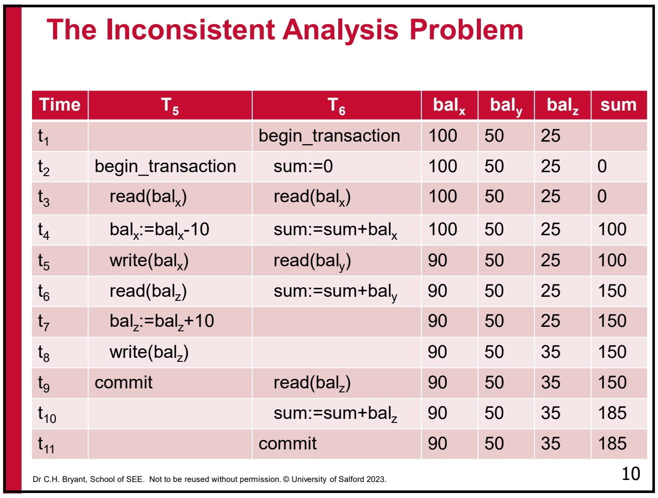
8



Summary of the Uncommitted Dependency Problem (Dirty Read)

- Occurs when one transaction can see intermediate results of another transaction before it has committed.
- T_4 updates bal_x to £200 but it aborts, so bal_x should be back at original value of £100.
- T_3 has read new value of bal_x (£200) and uses this value as the basis of £10 reduction, giving a new balance of £190, instead of £90.
- This problem could be avoided by preventing T_3 from reading bal_x until after T_4 commits or aborts.

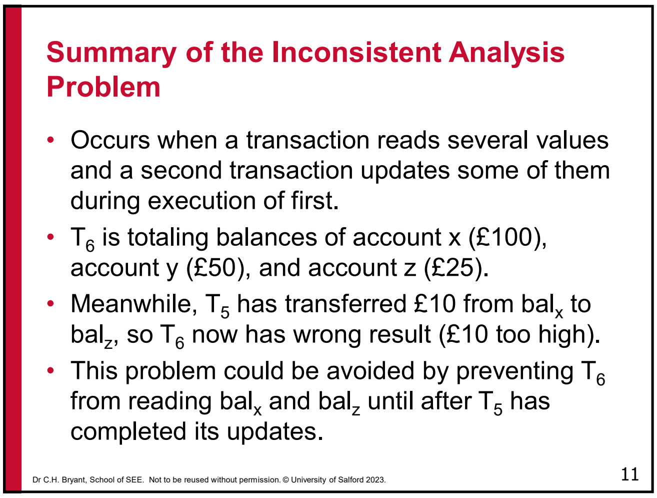
9



The Inconsistent Analysis Problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum:=0	100	50	25	0
t_3	read(bal_x)	read(bal_x)	100	50	25	0
t_4	bal_x:=bal_x-10	sum:=sum+bal_x	100	50	25	100
t_5	write(bal_x)	read(bal_y)	90	50	25	100
t_6	read(bal_z)	sum:=sum+bal_y	90	50	25	150
t_7	bal_z:=bal_z+10		90	50	25	150
t_8	write(bal_z)		90	50	35	150
t_9	commit	read(bal_z)	90	50	35	150
t_{10}		sum:=sum+bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

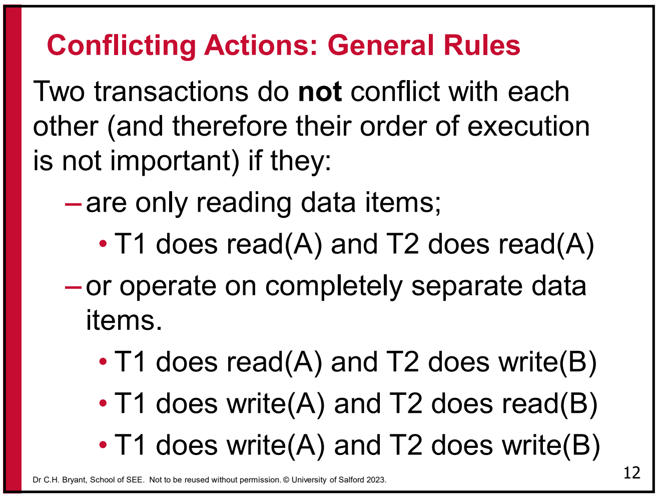
10



Summary of the Inconsistent Analysis Problem

- Occurs when a transaction reads several values and a second transaction updates some of them during execution of first.
- T_6 is totaling balances of account x (£100), account y (£50), and account z (£25).
- Meanwhile, T_5 has transferred £10 from bal_x to bal_z, so T_6 now has wrong result (£10 too high).
- This problem could be avoided by preventing T_6 from reading bal_x and bal_z until after T_5 has completed its updates.

11

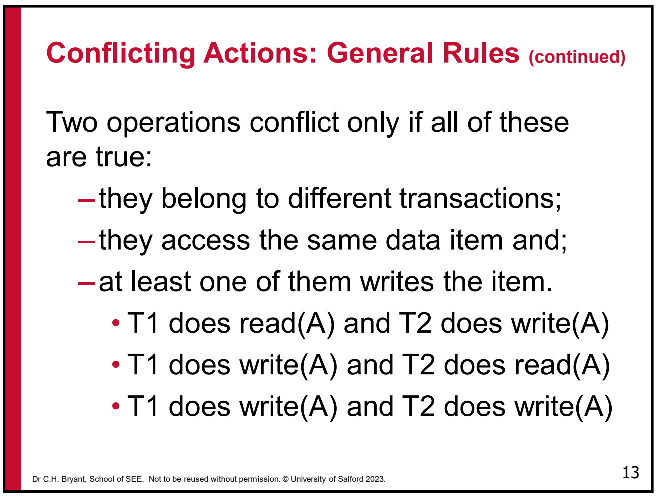


Conflicting Actions: General Rules

Two transactions do **not** conflict with each other (and therefore their order of execution is not important) if they:

- are only reading data items;
  - T1 does read(A) and T2 does read(A)
- or operate on completely separate data items.
  - T1 does read(A) and T2 does write(B)
  - T1 does write(A) and T2 does read(B)
  - T1 does write(A) and T2 does write(B)

12

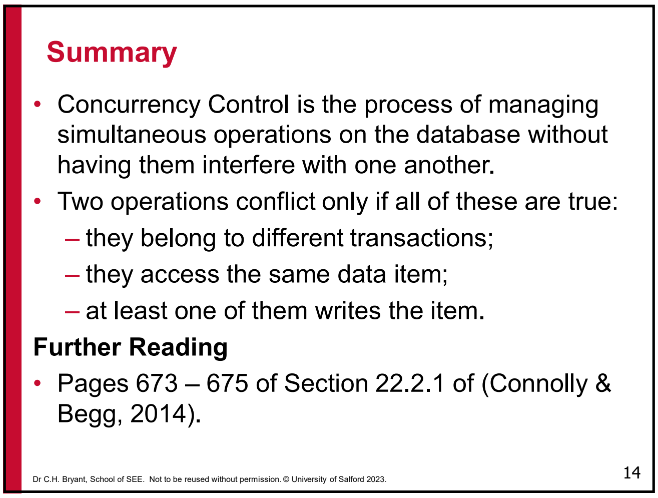


Conflicting Actions: General Rules (continued)

Two operations conflict only if all of these are true:

- they belong to different transactions;
- they access the same data item and;
- at least one of them writes the item.
  - T1 does read(A) and T2 does write(A)
  - T1 does write(A) and T2 does read(A)
  - T1 does write(A) and T2 does write(A)

13



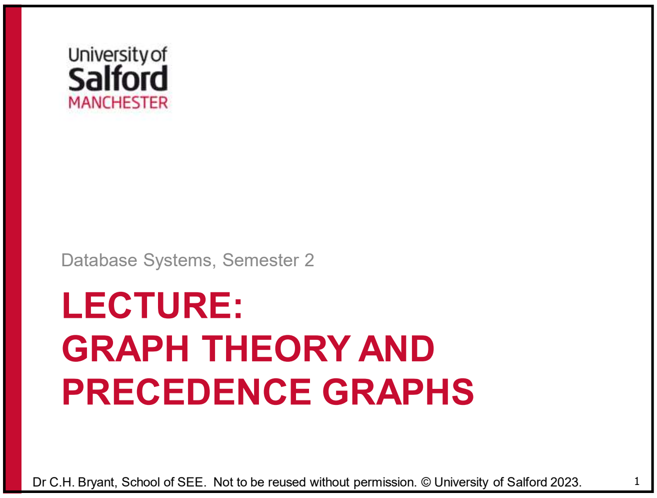
Summary

- Concurrency Control is the process of managing simultaneous operations on the database without having them interfere with one another.
- Two operations conflict only if all of these are true:
  - they belong to different transactions;
  - they access the same data item;
  - at least one of them writes the item.

Further Reading

- Pages 673 – 675 of Section 22.2.1 of (Connolly & Begg, 2014).

14



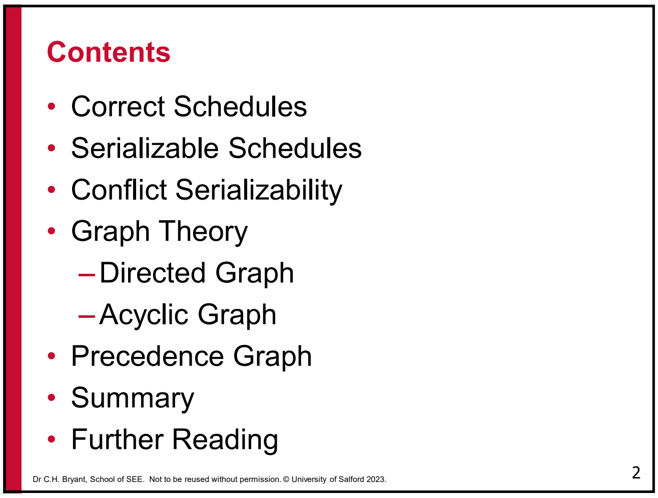
University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: GRAPH THEORY AND PRECEDENCE GRAPHS

1

1

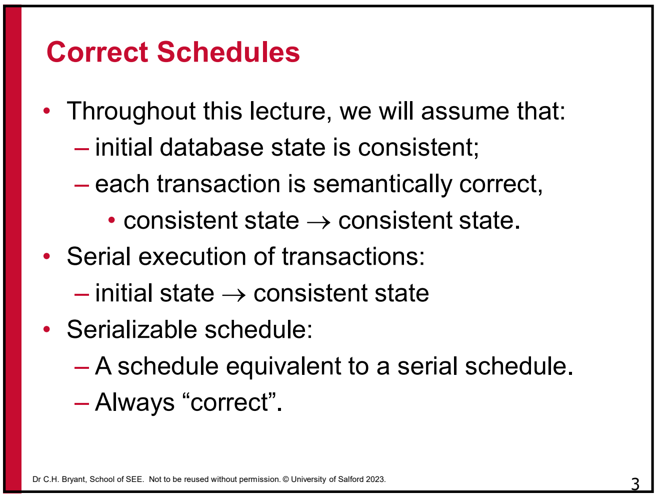


Contents

- Correct Schedules
- Serializable Schedules
- Conflict Serializability
- Graph Theory
  - Directed Graph
  - Acyclic Graph
- Precedence Graph
- Summary
- Further Reading

2

2

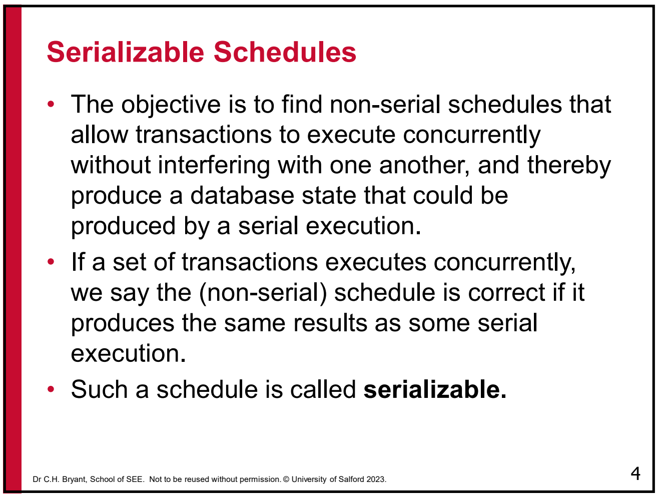


Correct Schedules

- Throughout this lecture, we will assume that:
  - initial database state is consistent;
  - each transaction is semantically correct,
    - consistent state → consistent state.
- Serial execution of transactions:
  - initial state → consistent state
- Serializable schedule:
  - A schedule equivalent to a serial schedule.
  - Always “correct”.

3

3

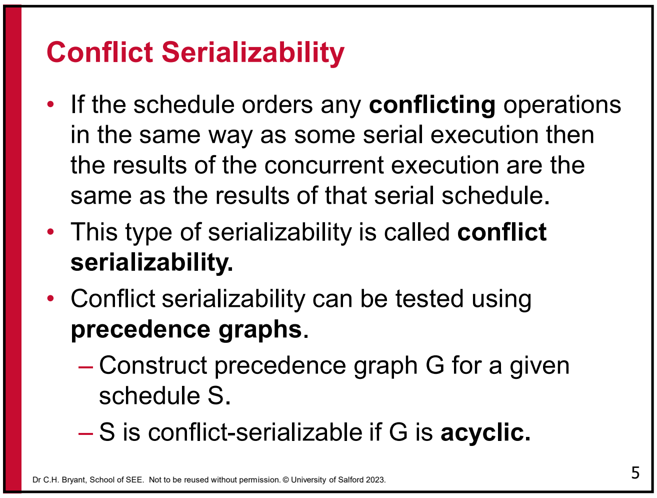


Serializable Schedules

- The objective is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- If a set of transactions executes concurrently, we say the (non-serial) schedule is correct if it produces the same results as some serial execution.
- Such a schedule is called **serializable.**

4

4

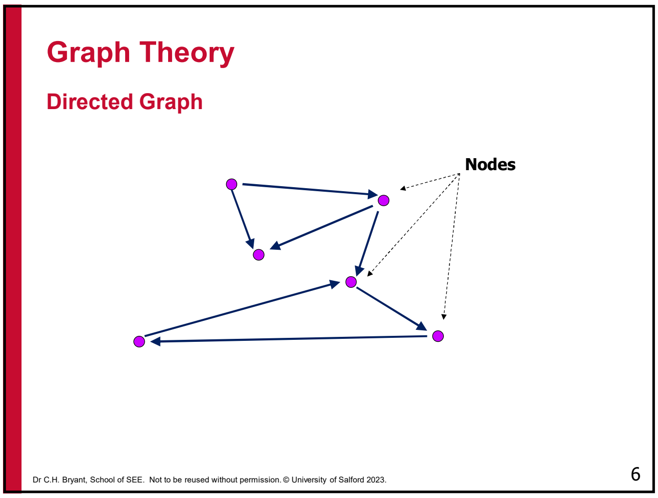


Conflict Serializability

- If the schedule orders any **conflicting** operations in the same way as some serial execution then the results of the concurrent execution are the same as the results of that serial schedule.
- This type of serializability is called **conflict serializability.**
- Conflict serializability can be tested using **precedence graphs**.
  - Construct precedence graph G for a given schedule S.
  - S is conflict-serializable if G is **acyclic.**

5

5

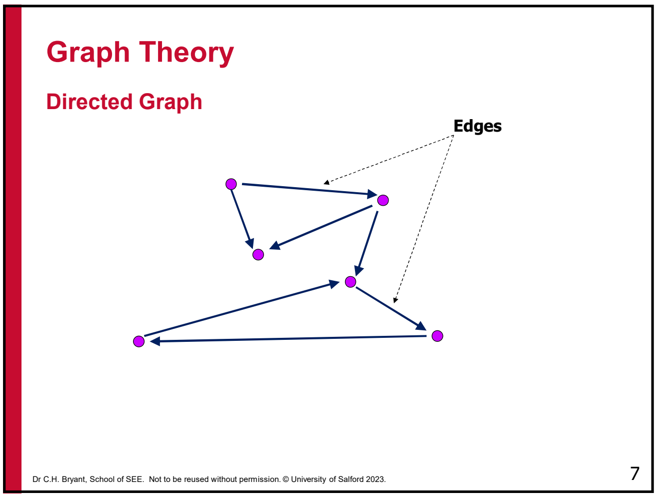


Graph Theory

Directed Graph

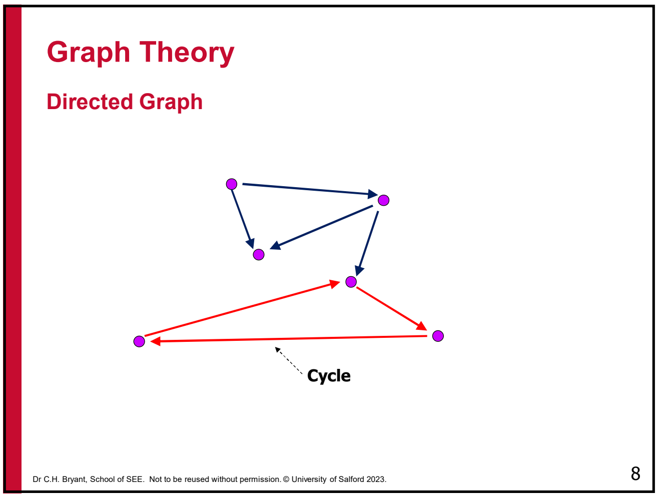
6

6



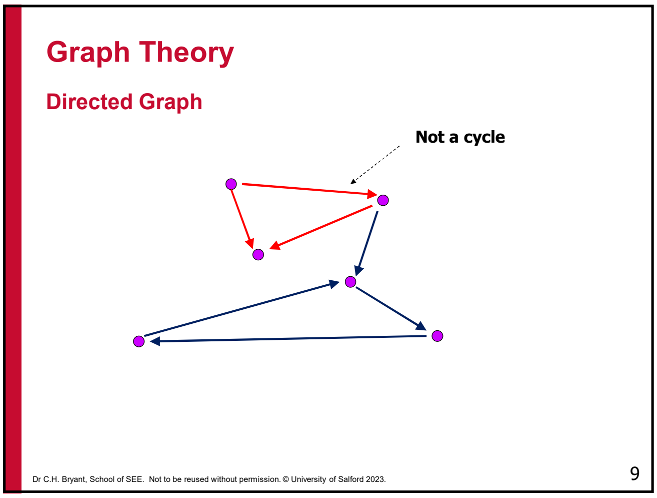
Graph Theory

Directed Graph



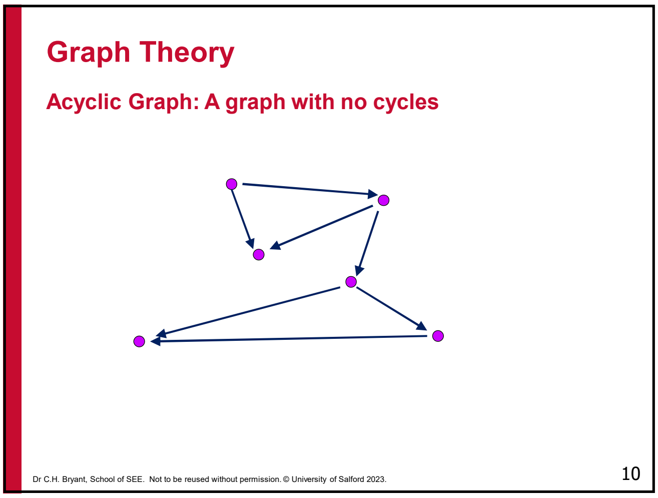
Graph Theory

Directed Graph



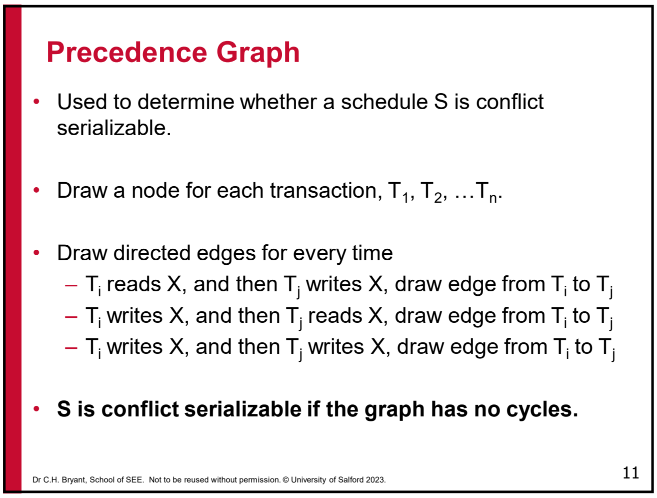
Graph Theory

Directed Graph



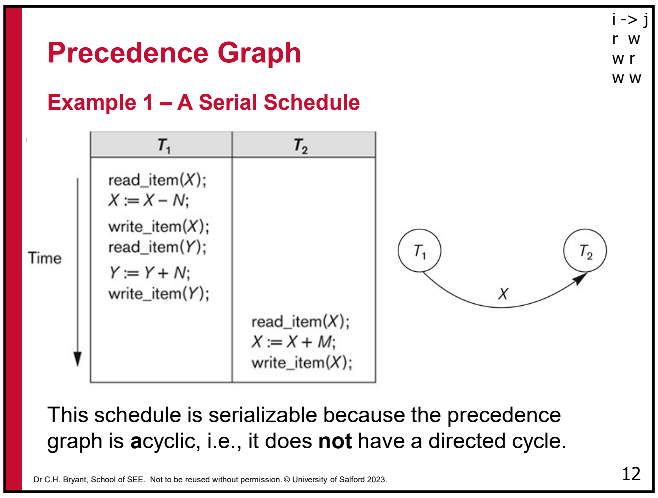
Graph Theory

Acyclic Graph: A graph with no cycles



Precedence Graph

- Used to determine whether a schedule S is conflict serializable.
- Draw a node for each transaction, T_1, T_2, …T_n.
- Draw directed edges for every time
  - T_i reads X, and then T_j writes X, draw edge from T_i to T_j
  - T_i writes X, and then T_j reads X, draw edge from T_i to T_j
  - T_i writes X, and then T_j writes X, draw edge from T_i to T_j
- **S is conflict serializable if the graph has no cycles.**



i -> j
r w
w r
w w

Precedence Graph

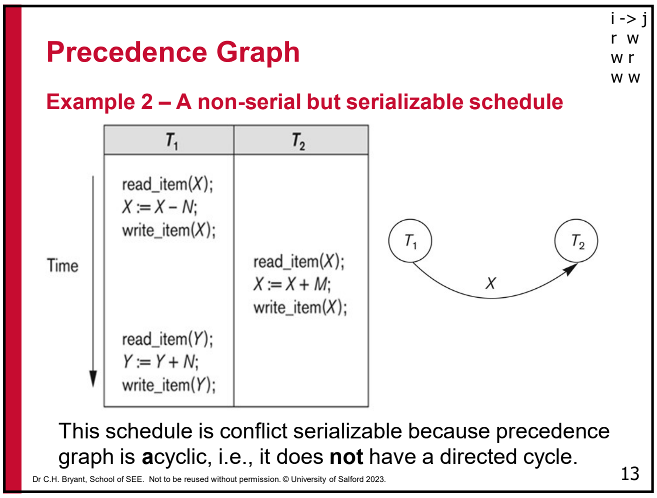
Example 1 – A Serial Schedule

T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);
read_item(Y);
$Y := Y + N$;
write_item(Y);	

	read_item(X);
	$X := X + M$;
	write_item(X);

This schedule is serializable because the precedence graph is **a**cyclic, i.e., it does **not** have a directed cycle.



i -> j
r w
w r
w w

Precedence Graph

Example 2 – A non-serial but serializable schedule

Time

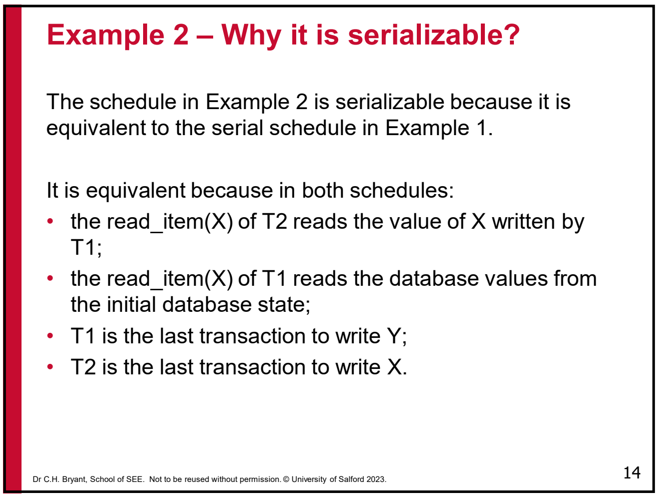
T_1	T_2

read_item(X);
$X := X - N$;
write_item(X);	

	read_item(X);
	$X := X + M$;
	write_item(X);

read_item(Y);
$Y := Y + N$;
write_item(Y);	

This schedule is conflict serializable because precedence graph is **a**cyclic, i.e., it does **not** have a directed cycle.

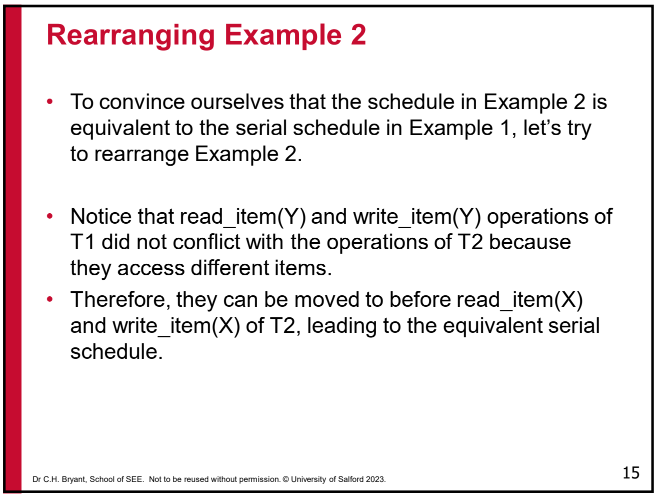


Example 2 – Why it is serializable?

The schedule in Example 2 is serializable because it is equivalent to the serial schedule in Example 1.

It is equivalent because in both schedules:

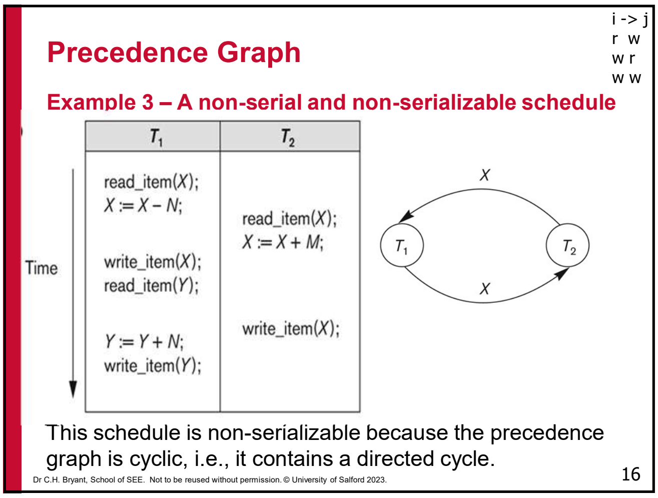
- the read_item(X) of T2 reads the value of X written by T1;
- the read_item(X) of T1 reads the database values from the initial database state;
- T1 is the last transaction to write Y;
- T2 is the last transaction to write X.



Rearranging Example 2

- To convince ourselves that the schedule in Example 2 is equivalent to the serial schedule in Example 1, let's try to rearrange Example 2.

- Notice that read_item(Y) and write_item(Y) operations of T1 did not conflict with the operations of T2 because they access different items.
- Therefore, they can be moved to before read_item(X) and write_item(X) of T2, leading to the equivalent serial schedule.



i -> j
r w
w r
w w

Precedence Graph

Example 3 – A non-serial and non-serializable schedule

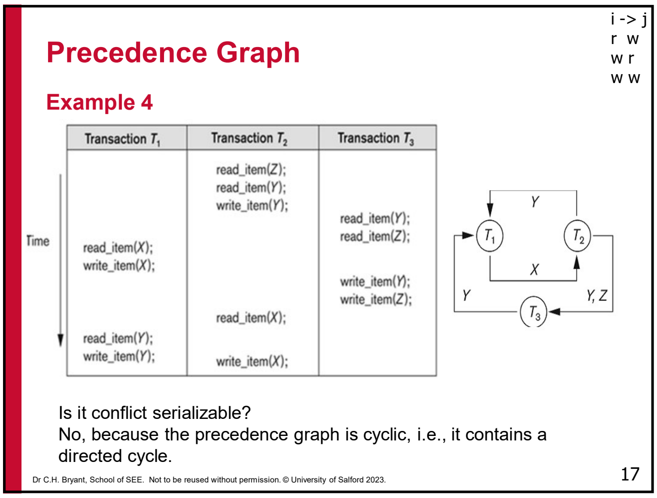
Time

T_1	T_2
read_item(X); $X := X - N$;	
	read_item(X); $X := X + M$;

write_item(X);
read_item(Y);	

	write_item(X);
$Y := Y + N$; write_item(Y);	

This schedule is non-serializable because the precedence graph is cyclic, i.e., it contains a directed cycle.



i -> j
r w
w r
w w

Precedence Graph

Example 4

Time

Transaction T_1	Transaction T_2	Transaction T_3

	read_item(Z);
	read_item(Y);
	write_item(Y);	

		read_item(Y);
		read_item(Z);

read_item(X);
write_item(X);		

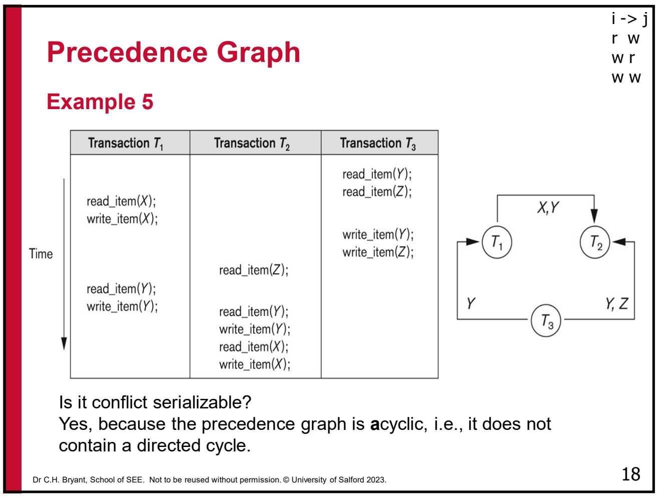
		write_item(Y);
		write_item(Z);

	read_item(X);	

read_item(Y);
write_item(Y);		

	write_item(X);	

Is it conflict serializable?
No, because the precedence graph is cyclic, i.e., it contains a directed cycle.



i -> j
r w
w r
w w

Precedence Graph

Example 5

Time

Transaction T_1	Transaction T_2	Transaction T_3

		read_item(Y);
		read_item(Z);

read_item(X);
write_item(X);		

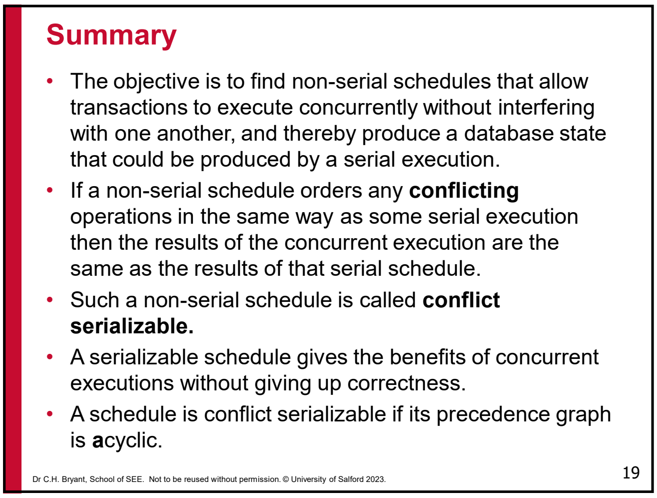
		write_item(Y);
		write_item(Z);

	read_item(Z);	

read_item(Y);
write_item(Y);		

	read_item(Y);
	write_item(Y);
	read_item(X);
	write_item(X);	

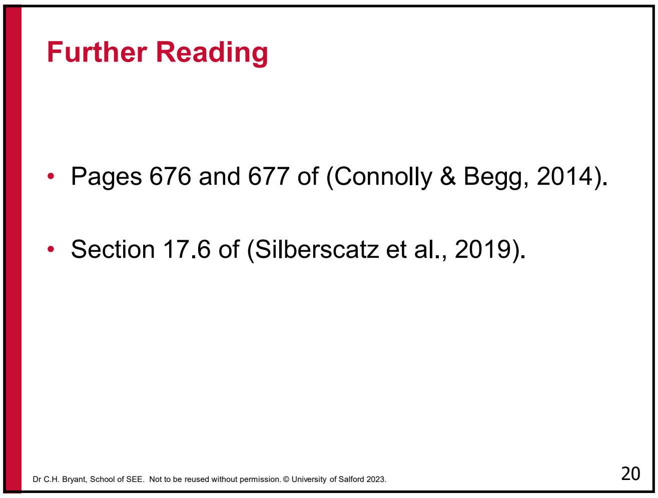
Is it conflict serializable?
Yes, because the precedence graph is **a**cyclic, i.e., it does not contain a directed cycle.



Summary

- The objective is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- If a non-serial schedule orders any **conflicting** operations in the same way as some serial execution then the results of the concurrent execution are the same as the results of that serial schedule.
- Such a non-serial schedule is called **conflict serializable.**
- A serializable schedule gives the benefits of concurrent executions without giving up correctness.
- A schedule is conflict serializable if its precedence graph is **a**cyclic.

19



Further Reading

- Pages 676 and 677 of (Connolly & Begg, 2014).
- Section 17.6 of (Silberscatz et al., 2019).

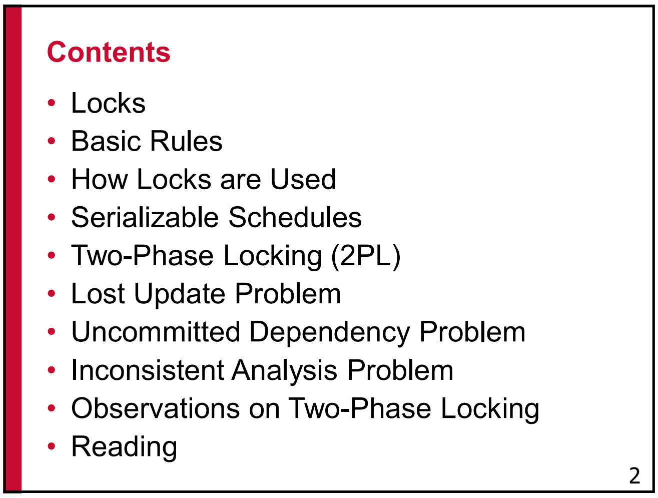
20

University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: LOCKS AND CONCURRENCY CONTROL

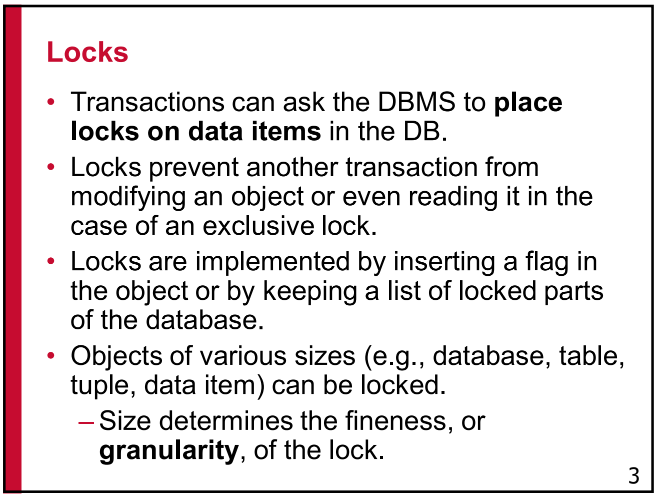
1



Contents

- Locks
- Basic Rules
- How Locks are Used
- Serializable Schedules
- Two-Phase Locking (2PL)
- Lost Update Problem
- Uncommitted Dependency Problem
- Inconsistent Analysis Problem
- Observations on Two-Phase Locking
- Reading

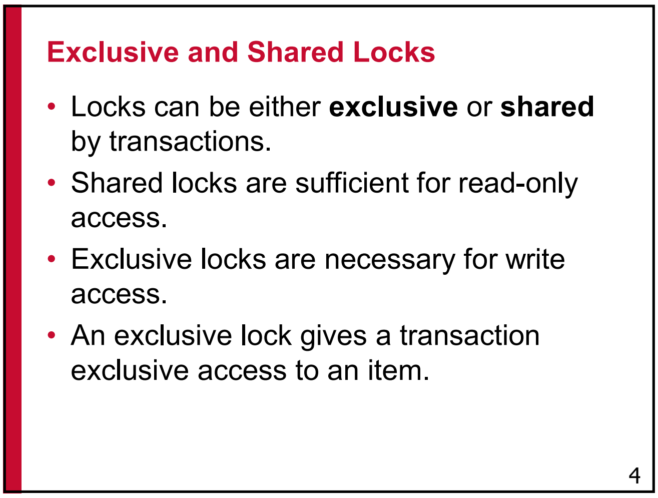
2



Locks

- Transactions can ask the DBMS to **place locks on data items** in the DB.
- Locks prevent another transaction from modifying an object or even reading it in the case of an exclusive lock.
- Locks are implemented by inserting a flag in the object or by keeping a list of locked parts of the database.
- Objects of various sizes (e.g., database, table, tuple, data item) can be locked.
  - Size determines the fineness, or **granularity**, of the lock.

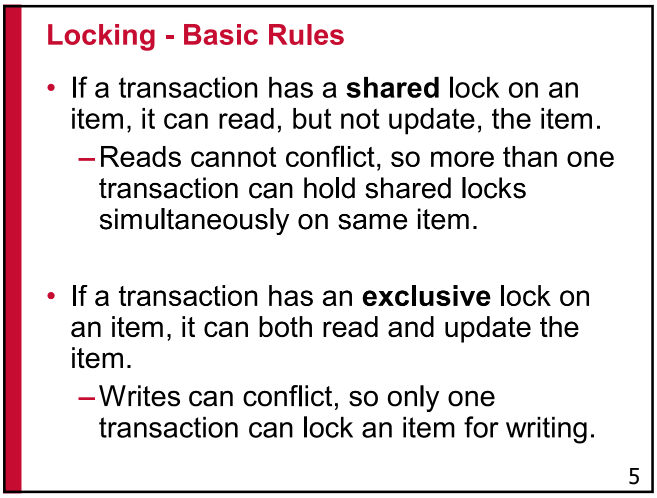
3



Exclusive and Shared Locks

- Locks can be either **exclusive** or **shared** by transactions.
- Shared locks are sufficient for read-only access.
- Exclusive locks are necessary for write access.
- An exclusive lock gives a transaction exclusive access to an item.

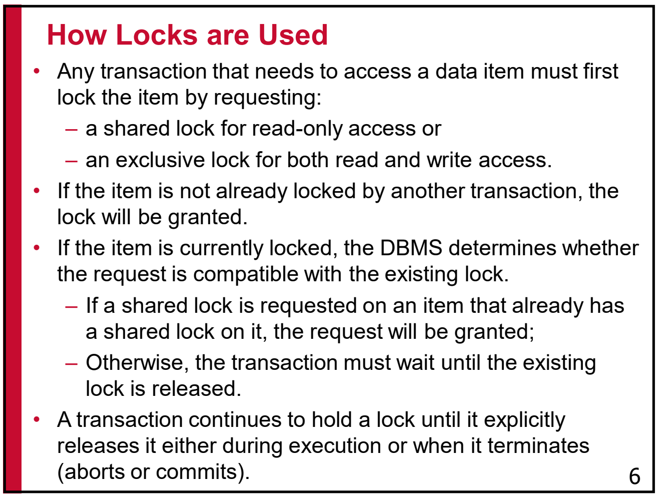
4



Locking - Basic Rules

- If a transaction has a **shared** lock on an item, it can read, but not update, the item.
  - Reads cannot conflict, so more than one transaction can hold shared locks simultaneously on same item.
- If a transaction has an **exclusive** lock on an item, it can both read and update the item.
  - Writes can conflict, so only one transaction can lock an item for writing.

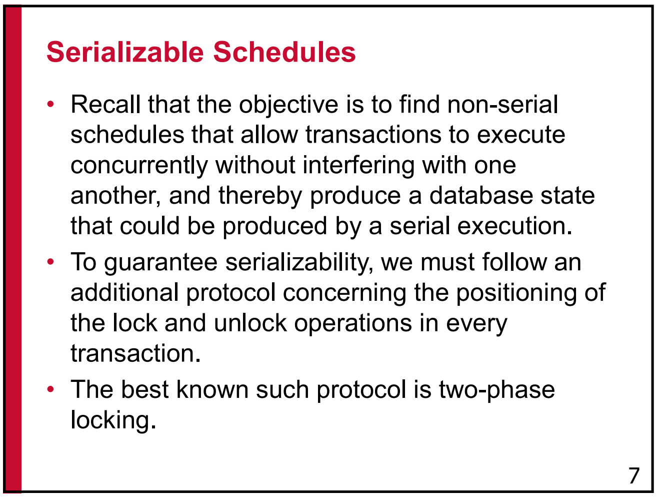
5



How Locks are Used

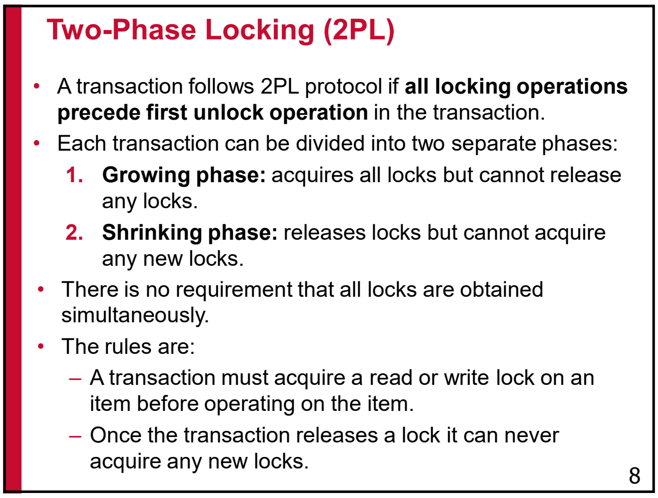
- Any transaction that needs to access a data item must first lock the item by requesting:
  - a shared lock for read-only access or
  - an exclusive lock for both read and write access.
- If the item is not already locked by another transaction, the lock will be granted.
- If the item is currently locked, the DBMS determines whether the request is compatible with the existing lock.
  - If a shared lock is requested on an item that already has a shared lock on it, the request will be granted;
  - Otherwise, the transaction must wait until the existing lock is released.
- A transaction continues to hold a lock until it explicitly releases it either during execution or when it terminates (aborts or commits).

6



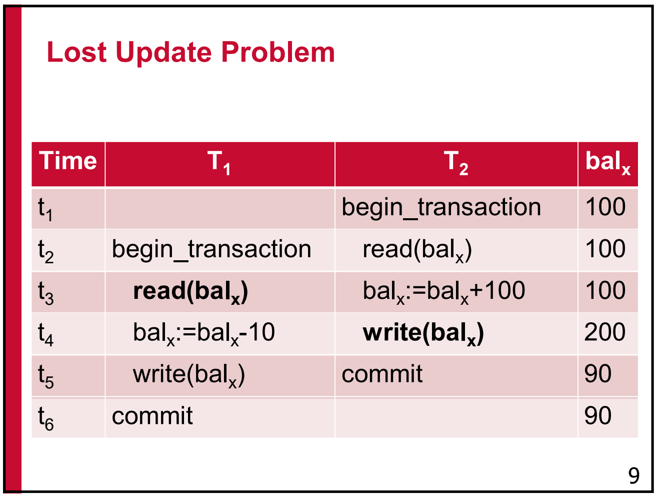
Serializable Schedules

- Recall that the objective is to find non-serial schedules that allow transactions to execute concurrently without interfering with one another, and thereby produce a database state that could be produced by a serial execution.
- To guarantee serializability, we must follow an additional protocol concerning the positioning of the lock and unlock operations in every transaction.
- The best known such protocol is two-phase locking.



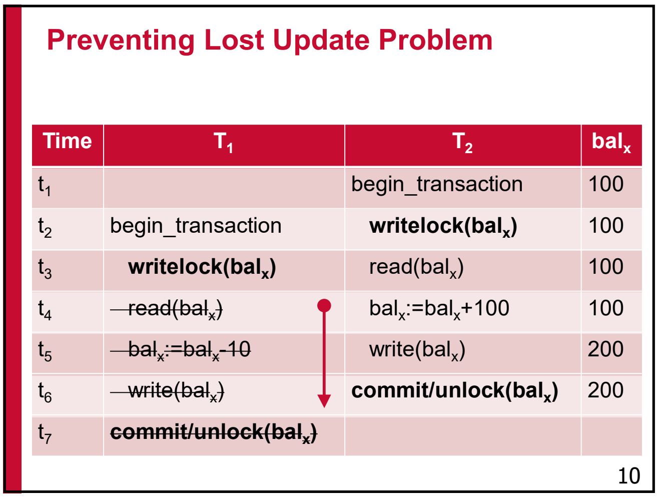
Two-Phase Locking (2PL)

- A transaction follows 2PL protocol if **all locking operations precede first unlock operation** in the transaction.
- Each transaction can be divided into two separate phases:
  1. **Growing phase:** acquires all locks but cannot release any locks.
  2. **Shrinking phase:** releases locks but cannot acquire any new locks.
- There is no requirement that all locks are obtained simultaneously.
- The rules are:
  - A transaction must acquire a read or write lock on an item before operating on the item.
  - Once the transaction releases a lock it can never acquire any new locks.



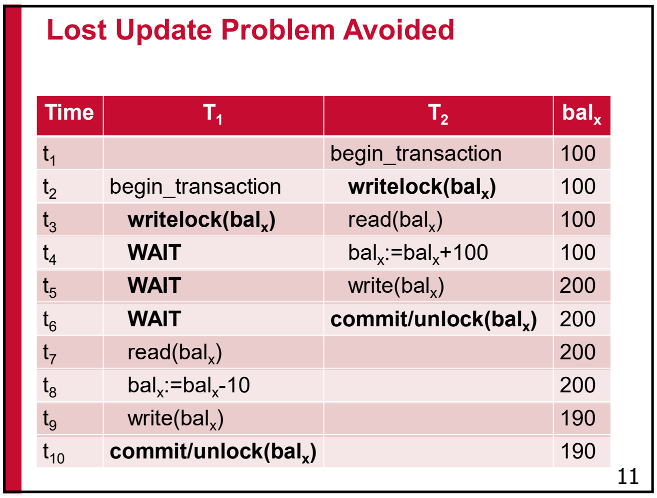
Lost Update Problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	read(bal_x)	100
t_3	**read(bal_x)**	bal_x:=bal_x+100	100
t_4	bal_x:=bal_x-10	**write(bal_x)**	200
t_5	write(bal_x)	commit	90
t_6	commit		90



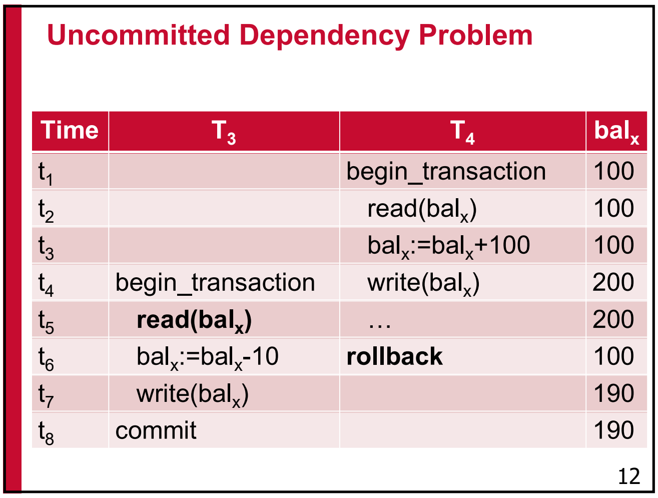
Preventing Lost Update Problem

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	**writelock(bal_x)**	100
t_3	**writelock(bal_x)**	read(bal_x)	100
t_4	~~read(bal_x)~~	bal_x:=bal_x+100	100
t_5	~~bal_x:=bal_x-10~~	write(bal_x)	200
t_6	~~write(bal_x)~~	**commit/unlock(bal_x)**	200
t_7	**~~commit/unlock(bal_x)~~**		



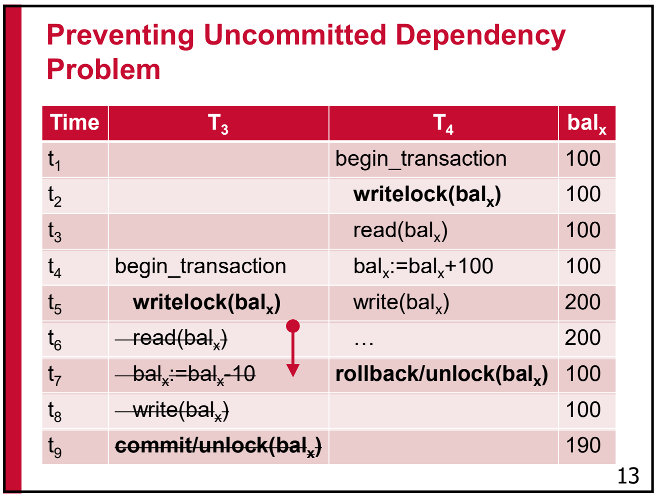
Lost Update Problem Avoided

Time	T_1	T_2	bal_x
t_1		begin_transaction	100
t_2	begin_transaction	**writelock(bal_x)**	100
t_3	**writelock(bal_x)**	read(bal_x)	100
t_4	**WAIT**	bal_x:=bal_x+100	100
t_5	**WAIT**	write(bal_x)	200
t_6	**WAIT**	**commit/unlock(bal_x)**	200
t_7	read(bal_x)		200
t_8	bal_x:=bal_x-10		200
t_9	write(bal_x)		190
t_{10}	**commit/unlock(bal_x)**		190



Uncommitted Dependency Problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		read(bal_x)	100
t_3		bal_x:=bal_x+100	100
t_4	begin_transaction	write(bal_x)	200
t_5	**read(bal_x)**	…	200
t_6	bal_x:=bal_x-10	**rollback**	100
t_7	write(bal_x)		190
t_8	commit		190

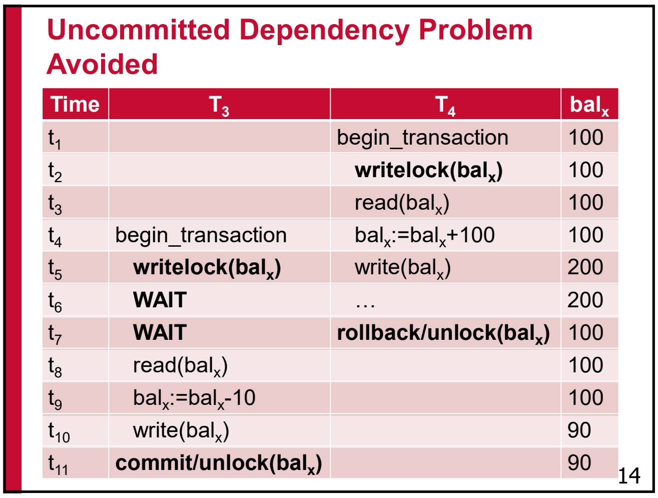


Preventing Uncommitted Dependency Problem

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		**writelock(bal_x)**	100
t_3		read(bal_x)	100
t_4	begin_transaction	bal_x:=bal_x+100	100
t_5	**writelock(bal_x)**	write(bal_x)	200
t_6	~~read(bal_x)~~	…	200
t_7	~~bal_x:=bal_x-10~~	**rollback/unlock(bal_x)**	100
t_8	~~write(bal_x)~~		100
t_9	~~**commit/unlock(bal_x)**~~		190

13

13

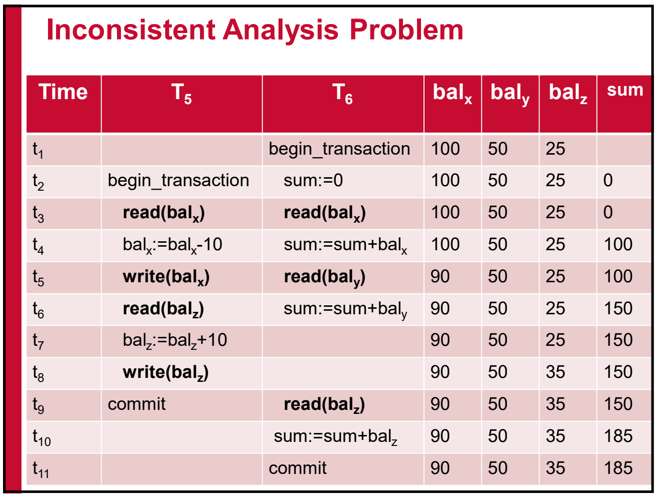


Uncommitted Dependency Problem Avoided

Time	T_3	T_4	bal_x
t_1		begin_transaction	100
t_2		**writelock(bal_x)**	100
t_3		read(bal_x)	100
t_4	begin_transaction	bal_x:=bal_x+100	100
t_5	**writelock(bal_x)**	write(bal_x)	200
t_6	**WAIT**	…	200
t_7	**WAIT**	**rollback/unlock(bal_x)**	100
t_8	read(bal_x)		100
t_9	bal_x:=bal_x-10		100
t_{10}	write(bal_x)		90
t_{11}	**commit/unlock(bal_x)**		90

14

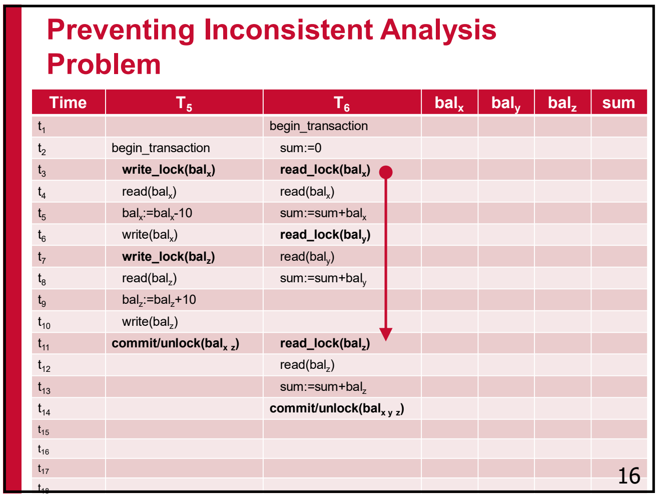
14



Inconsistent Analysis Problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum:=0	100	50	25	0
t_3	**read(bal_x)**	**read(bal_x)**	100	50	25	0
t_4	bal_x:=bal_x-10	sum:=sum+bal_x	100	50	25	100
t_5	**write(bal_x)**	**read(bal_y)**	90	50	25	100
t_6	**read(bal_z)**	sum:=sum+bal_y	90	50	25	150
t_7	bal_z:=bal_z+10		90	50	25	150
t_8	**write(bal_z)**		90	50	35	150
t_9	commit	**read(bal_z)**	90	50	35	150
t_{10}		sum:=sum+bal_z	90	50	35	185
t_{11}		commit	90	50	35	185

15

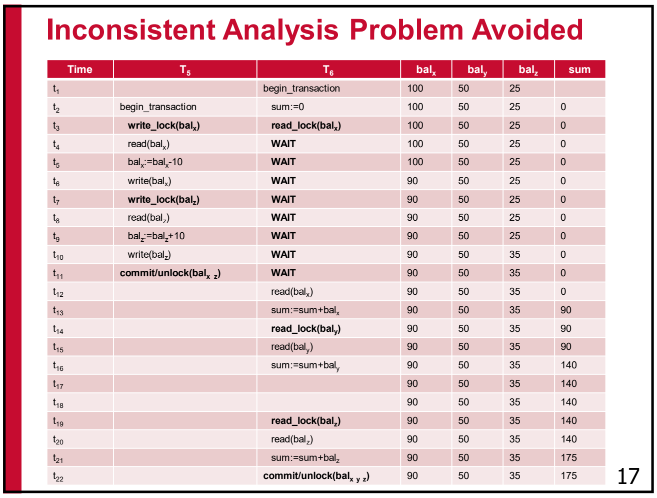


Preventing Inconsistent Analysis Problem

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction				
t_2	begin_transaction	sum:=0				
t_3	**write_lock(bal_x)**	**read_lock(bal_x)**				
t_4	read(bal_x)	read(bal_x)				
t_5	bal_x:=bal_x-10	sum:=sum+bal_x				
t_6	write(bal_x)	**read_lock(bal_y)**				
t_7	**write_lock(bal_z)**	read(bal_y)				
t_8	read(bal_z)	sum:=sum+bal_y				
t_9	bal_z:=bal_z+10					
t_{10}	write(bal_z)					
t_{11}	**commit/unlock($bal_{x\ z}$)**	**read_lock(bal_z)**				
t_{12}		read(bal_z)				
t_{13}		sum:=sum+bal_z				
t_{14}		**commit/unlock($bal_{x\ y\ z}$)**				
t_{15}						
t_{16}						
t_{17}						
t_{18}						

16

16

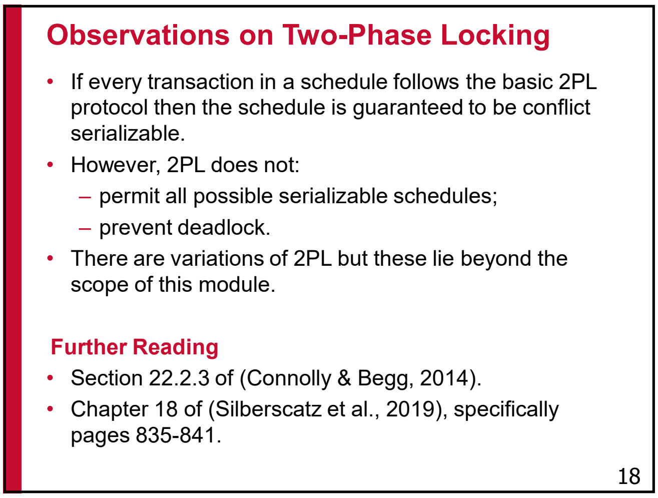


Inconsistent Analysis Problem Avoided

Time	T_5	T_6	bal_x	bal_y	bal_z	sum
t_1		begin_transaction	100	50	25	
t_2	begin_transaction	sum:=0	100	50	25	0
t_3	**write_lock(bal_x)**	**read_lock(bal_x)**	100	50	25	0
t_4	read(bal_x)	**WAIT**	100	50	25	0
t_5	bal_x:=bal_x-10	**WAIT**	100	50	25	0
t_6	write(bal_x)	**WAIT**	90	50	25	0
t_7	**write_lock(bal_z)**	**WAIT**	90	50	25	0
t_8	read(bal_z)	**WAIT**	90	50	25	0
t_9	bal_z:=bal_z+10	**WAIT**	90	50	25	0
t_{10}	write(bal_z)	**WAIT**	90	50	35	0
t_{11}	**commit/unlock($bal_{x\ z}$)**	**WAIT**	90	50	35	0
t_{12}		read(bal_x)	90	50	35	0
t_{13}		sum:=sum+bal_x	90	50	35	90
t_{14}		**read_lock(bal_y)**	90	50	35	90
t_{15}		read(bal_y)	90	50	35	90
t_{16}		sum:=sum+bal_y	90	50	35	140
t_{17}			90	50	35	140
t_{18}			90	50	35	140
t_{19}		**read_lock(bal_z)**	90	50	35	140
t_{20}		read(bal_z)	90	50	35	140
t_{21}		sum:=sum+bal_z	90	50	35	175
t_{22}		**commit/unlock($bal_{x\ y\ z}$)**	90	50	35	175

17

17



Observations on Two-Phase Locking

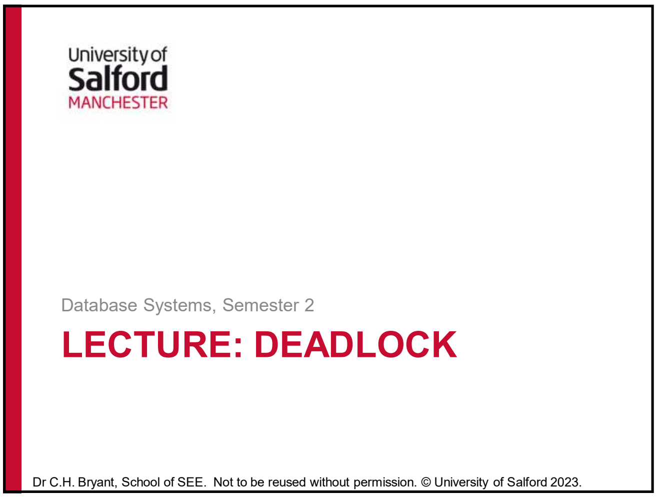
- If every transaction in a schedule follows the basic 2PL protocol then the schedule is guaranteed to be conflict serializable.
- However, 2PL does not:
  - permit all possible serializable schedules;
  - prevent deadlock.
- There are variations of 2PL but these lie beyond the scope of this module.

Further Reading

- Section 22.2.3 of (Connolly & Begg, 2014).
- Chapter 18 of (Silberscatz et al., 2019), specifically pages 835-841.

18

18

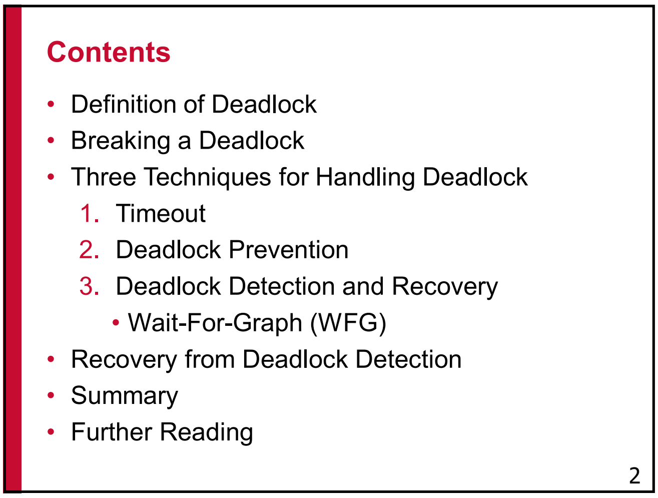


University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: DEADLOCK

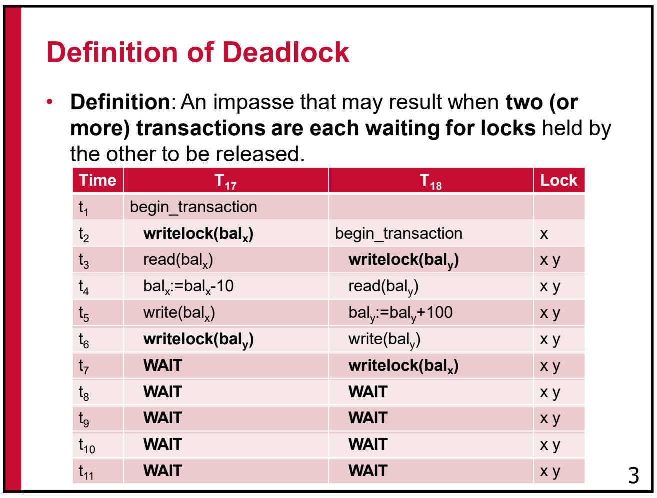
1



Contents

- Definition of Deadlock
- Breaking a Deadlock
- Three Techniques for Handling Deadlock
  1. Timeout
  2. Deadlock Prevention
  3. Deadlock Detection and Recovery
     - Wait-For-Graph (WFG)
- Recovery from Deadlock Detection
- Summary
- Further Reading

2

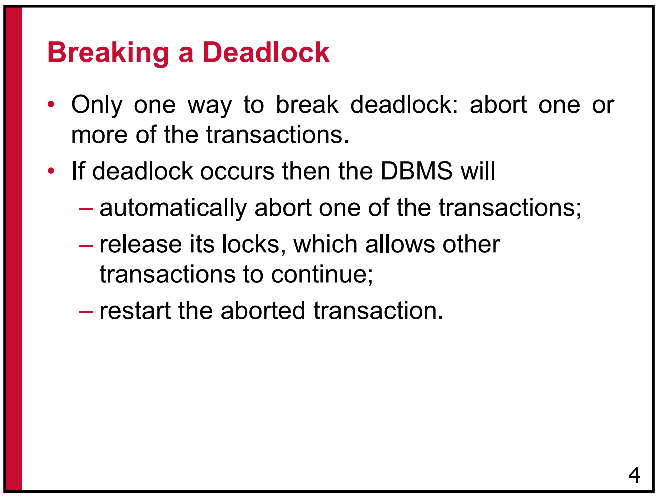


Definition of Deadlock

- **Definition**: An impasse that may result when **two (or more) transactions are each waiting for locks** held by the other to be released.

Time	T_{17}	T_{18}	Lock
t_1	begin_transaction		
t_2	**writelock(bal_x)**	begin_transaction	x
t_3	read(bal_x)	**writelock(bal_y)**	x y
t_4	bal_x:=bal_x-10	read(bal_y)	x y
t_5	write(bal_x)	bal_y:=bal_y+100	x y
t_6	**writelock(bal_y)**	write(bal_y)	x y
t_7	**WAIT**	**writelock(bal_x)**	x y
t_8	**WAIT**	**WAIT**	x y
t_9	**WAIT**	**WAIT**	x y
t_{10}	**WAIT**	**WAIT**	x y
t_{11}	**WAIT**	**WAIT**	x y

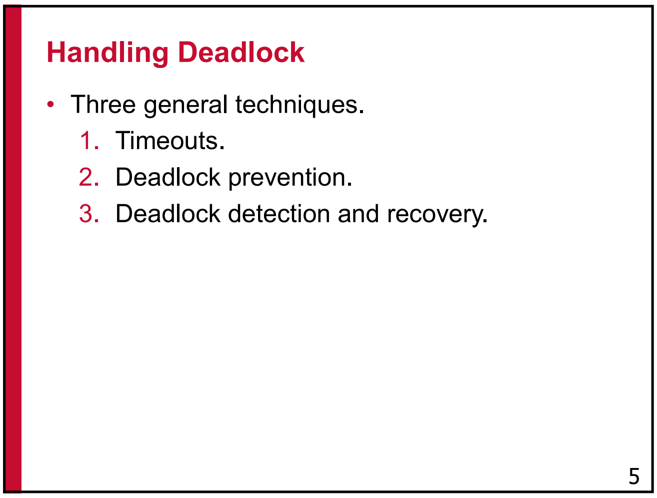
3



Breaking a Deadlock

- Only one way to break deadlock: abort one or more of the transactions.
- If deadlock occurs then the DBMS will
  - automatically abort one of the transactions;
  - release its locks, which allows other transactions to continue;
  - restart the aborted transaction.

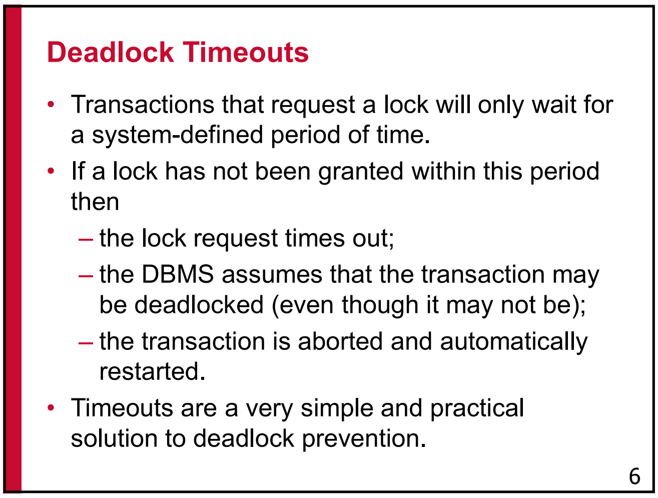
4



Handling Deadlock

- Three general techniques.
  1. Timeouts.
  2. Deadlock prevention.
  3. Deadlock detection and recovery.

5



Deadlock Timeouts

- Transactions that request a lock will only wait for a system-defined period of time.
- If a lock has not been granted within this period then
  - the lock request times out;
  - the DBMS assumes that the transaction may be deadlocked (even though it may not be);
  - the transaction is aborted and automatically restarted.
- Timeouts are a very simple and practical solution to deadlock prevention.

6

Deadlock Prevention

- DBMS **looks ahead** to see if a transaction would cause deadlock and prevents deadlock from occurring.
- Could order transactions using transaction timestamps:
  - **Wait-Die: only an older transaction can wait for a younger one.**
    - If a younger transaction requests a lock held by an older one, then the younger one is aborted (dies) and restarted with same timestamp, so it will eventually become the oldest active transaction and will not die.
  - **Wound-Wait: only a younger transaction can wait for an older one**.
    - If an older transaction requests a lock held by a younger one, then the younger one is aborted (wounded).

Deadlock Prevention (continued)

Wait-Die vs **Wound-Wait**

Holding	Requests	Result
Younger	Older	Older waits
Older	Younger	Younger aborts (dies)

Holding	Requests	Result
Younger	**Older**	Younger aborts (wounded)
Older	**Younger**	Younger waits

Both schemes abort the younger of the two transactions that may be involved in deadlock on the basis that this will waste less processing.

Conclusion on wait-die and wound-wait

- These techniques are deadlock free.
- In wait-die, transactions only wait for younger transactions, so no cycle is created.
- In wound-wait, transactions only wait for older transactions, so no cycle is created.
- However, both may cause some transactions to be aborted and restarted needlessly, even though those transactions may never actually cause a deadlock.

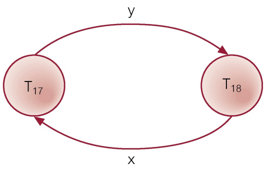
Deadlock Detection and Recovery

- DBMS allows deadlock to occur but then recognizes it and breaks it.
- The DBMS usually uses a **wait-for graph** (WFG) that shows transaction dependencies.
- A WFT is generated by creating:
  - a node for each transaction;
  - an edge T_i -> T_j, if T_i is waiting to lock an item locked by T_j.
- Deadlock exists if, and only if, the WFG contains a cycle.
- A WFG is created at regular intervals.

Wait-For-Graph (WFG) – Example One

Time	T_{17}	T_{18}
t_1	begin_transaction	
t_2	**writelock(bal_x)**	begin_transaction
t_3	read(bal_x)	**writelock(bal_y)**
t_4	bal_x:=bal_x-10	read(bal_y)
t_5	write(bal_x)	bal_y:=bal_y+100
t_6	**writelock(bal_y)**	write(bal_y)
t_7	**WAIT**	**writelock(bal_x)**
t_8	**WAIT**	**WAIT**
t_9	**WAIT**	**WAIT**
t_{10}	**WAIT**	**WAIT**
t_{11}	**WAIT**	**WAIT**

Trans-action	Data items locked by transaction	Data items is waiting for
T_{17}	X	Y
T_{18}	Y	X



edge T_i -> T_j, if T_i waiting to lock item locked by T_j

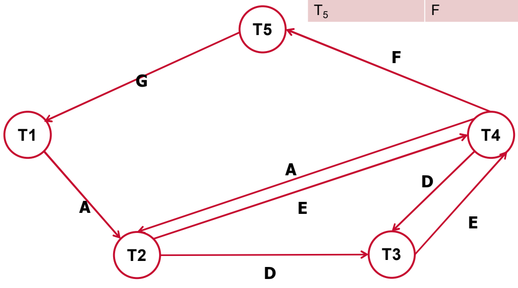
Example Two

Transaction	Data items locked by transaction	Data items is waiting for
T_1	G	A
T_2	A	D E
T_3	D	E
T_4	E	F D A
T_5	F	G

edge T_i -> T_j, if T_i waiting to lock item locked by T_j

WFG for Example Two

Transaction	Data items locked by transaction	Data items is waiting for
T_1	G	A
T_2	A	D E
T_3	D	E
T_4	E	F D A
T_5	F	G



Deadlock exists because the WFG contains a cycle.

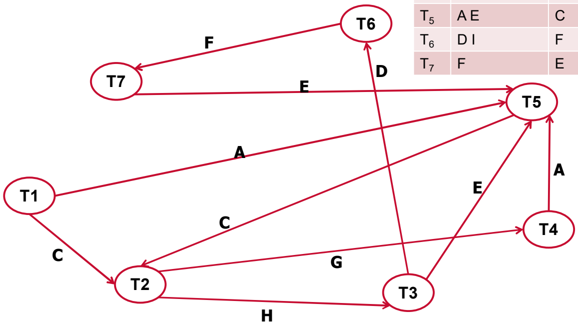
Example Three

Transaction	Data items locked by transaction	Data items is waiting for
T_1	B	A C
T_2	C J	G H
T_3	H	D E
T_4	G	A
T_5	A E	C
T_6	D I	F
T_7	F	E

edge T_i -> T_j, if T_i waiting to lock item locked by T_j

WFG for Example Three

T	Data items locked by T	Data items waiting for
T_1	B	A C
T_2	C J	G H
T_3	H	D E
T_4	G	A
T_5	A E	C
T_6	D I	F
T_7	F	E



Deadlock exists because the WFG contains a cycle.

Recovery from Deadlock Detection

There are several issues that the DBMS needs to address.

- Selecting a deadlock victim that will minimize the cost of breaking the deadlock.
  - It may be better to abort a transaction that has just started, rather than one that been running for a long time.
  - It may be better to abort a transaction that has made little change to the database, rather than one that has made significant change.

Recovery from Deadlock Detection (continued)

- Avoiding starvation.
  - This occurs when the same transaction is always chosen as the victim, preventing it from ever completing.
  - The DBMS may try to avoid this by storing a count of the number of times a transaction has been aborted.

Summary

- Deadlock is an impasse that may result when **two (or more) transactions are each waiting for locks** held by the other to be released.
- There are three general techniques for handling deadlock:
  1. Timeouts - simple and practical.
  2. Deadlock prevention - may cause some transactions to be aborted and restarted needlessly.
  3. Deadlock detection (using WFG) and recovery.

Further Reading

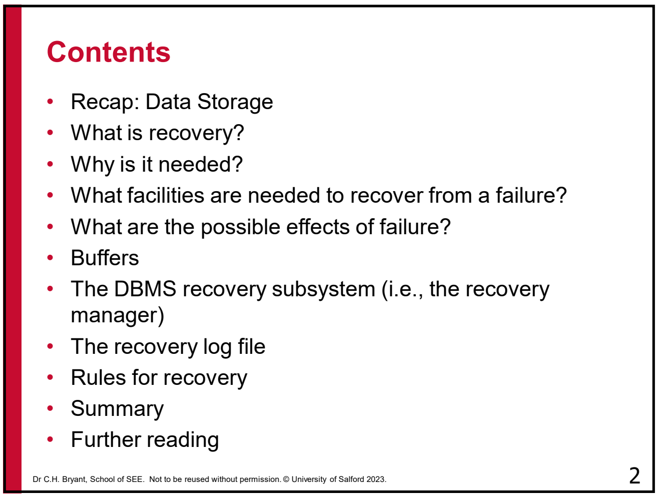
- Section 22.2.4 of (Connolly & Begg, 2014).
- Section 18.2 of (Silberscatz et al., 2019).

University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: DATABASE RECOVERY: PART 1

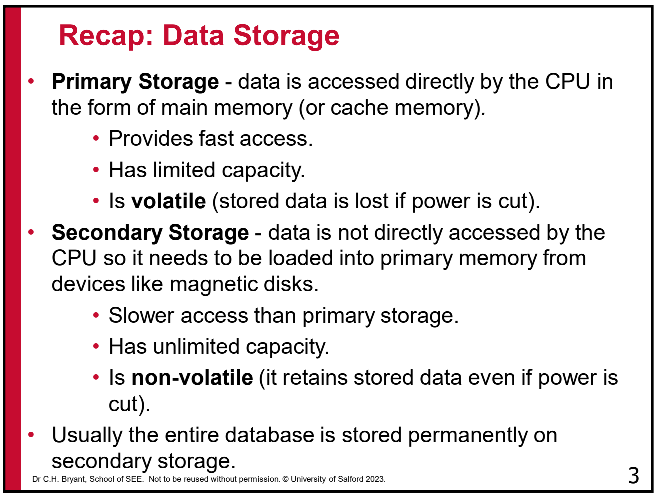
1



Contents

- Recap: Data Storage
- What is recovery?
- Why is it needed?
- What facilities are needed to recover from a failure?
- What are the possible effects of failure?
- Buffers
- The DBMS recovery subsystem (i.e., the recovery manager)
- The recovery log file
- Rules for recovery
- Summary
- Further reading

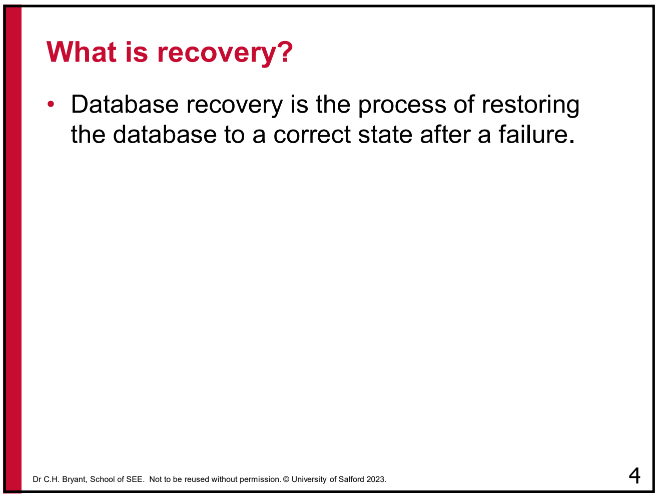
2



Recap: Data Storage

- **Primary Storage** - data is accessed directly by the CPU in the form of main memory (or cache memory).
  - Provides fast access.
  - Has limited capacity.
  - Is **volatile** (stored data is lost if power is cut).
- **Secondary Storage** - data is not directly accessed by the CPU so it needs to be loaded into primary memory from devices like magnetic disks.
  - Slower access than primary storage.
  - Has unlimited capacity.
  - Is **non-volatile** (it retains stored data even if power is cut).
- Usually the entire database is stored permanently on secondary storage.

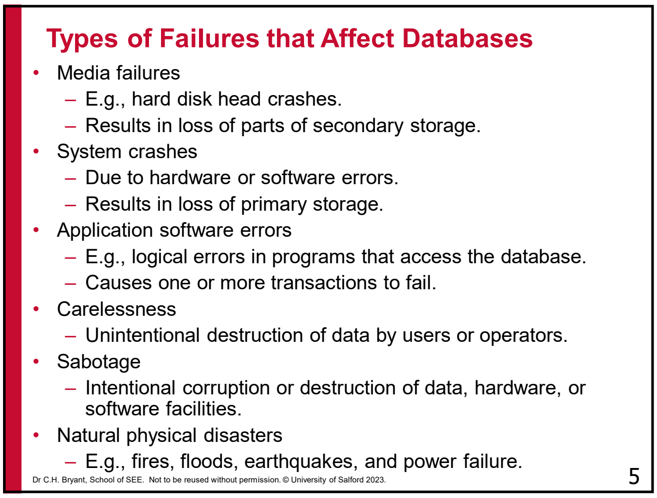
3



What is recovery?

- Database recovery is the process of restoring the database to a correct state after a failure.

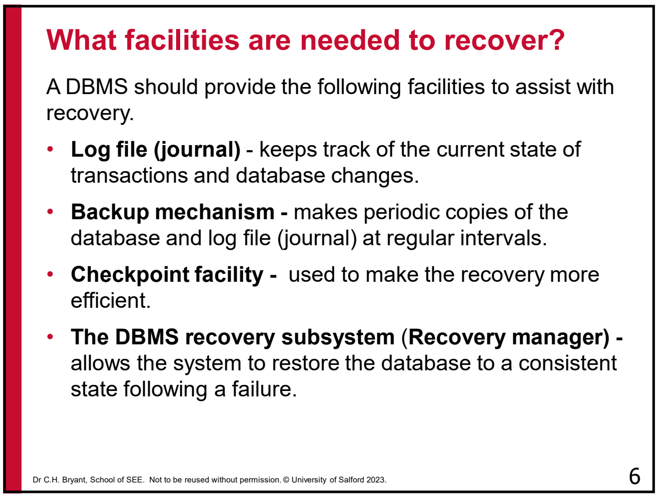
4



Types of Failures that Affect Databases

- Media failures
  - E.g., hard disk head crashes.
  - Results in loss of parts of secondary storage.
- System crashes
  - Due to hardware or software errors.
  - Results in loss of primary storage.
- Application software errors
  - E.g., logical errors in programs that access the database.
  - Causes one or more transactions to fail.
- Carelessness
  - Unintentional destruction of data by users or operators.
- Sabotage
  - Intentional corruption or destruction of data, hardware, or software facilities.
- Natural physical disasters
  - E.g., fires, floods, earthquakes, and power failure.

5

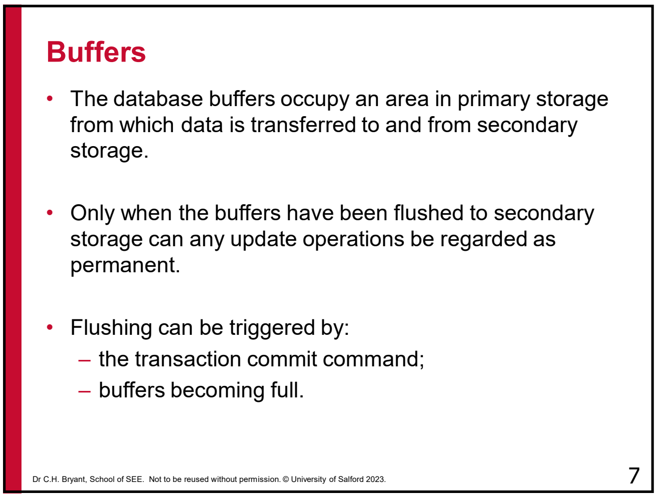


What facilities are needed to recover?

A DBMS should provide the following facilities to assist with recovery.

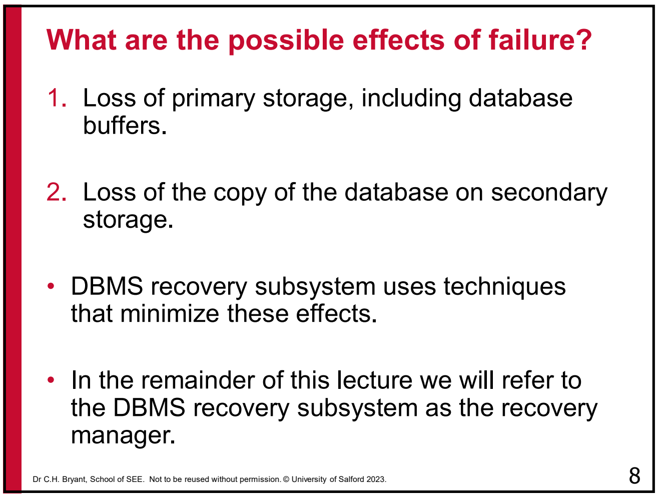
- **Log file (journal)** - keeps track of the current state of transactions and database changes.
- **Backup mechanism -** makes periodic copies of the database and log file (journal) at regular intervals.
- **Checkpoint facility -** used to make the recovery more efficient.
- **The DBMS recovery subsystem** (**Recovery manager) -** allows the system to restore the database to a consistent state following a failure.

6



Buffers

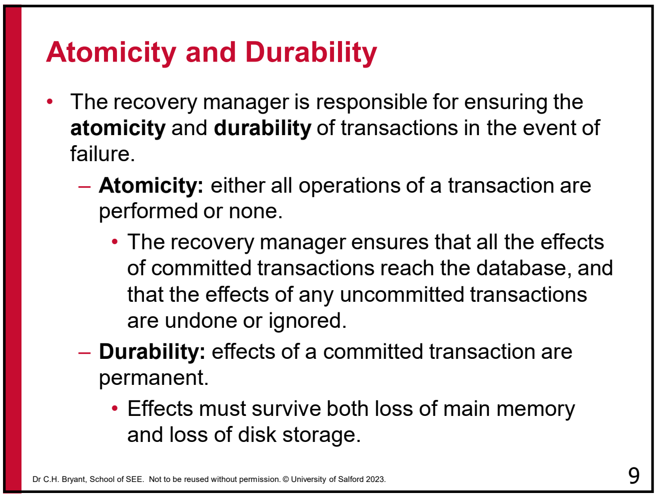
- The database buffers occupy an area in primary storage from which data is transferred to and from secondary storage.
- Only when the buffers have been flushed to secondary storage can any update operations be regarded as permanent.
- Flushing can be triggered by:
  - the transaction commit command;
  - buffers becoming full.



What are the possible effects of failure?

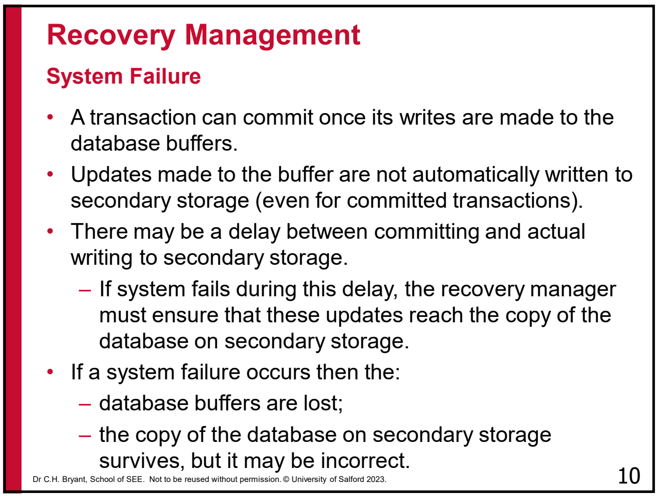
1. Loss of primary storage, including database buffers.
2. Loss of the copy of the database on secondary storage.

- DBMS recovery subsystem uses techniques that minimize these effects.
- In the remainder of this lecture we will refer to the DBMS recovery subsystem as the recovery manager.



Atomicity and Durability

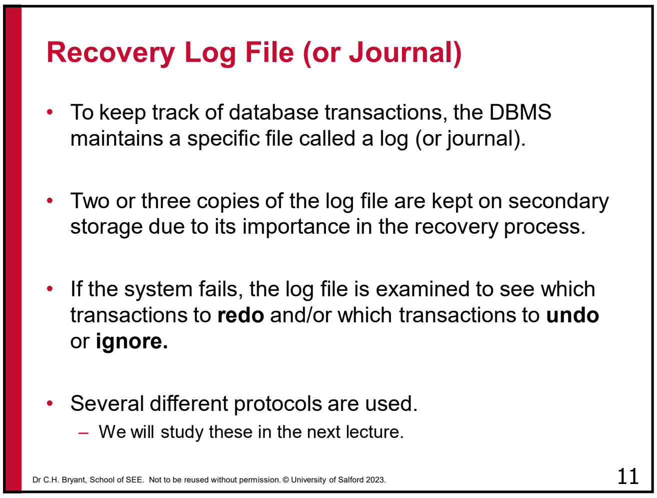
- The recovery manager is responsible for ensuring the **atomicity** and **durability** of transactions in the event of failure.
  - **Atomicity:** either all operations of a transaction are performed or none.
    - The recovery manager ensures that all the effects of committed transactions reach the database, and that the effects of any uncommitted transactions are undone or ignored.
  - **Durability:** effects of a committed transaction are permanent.
    - Effects must survive both loss of main memory and loss of disk storage.



Recovery Management

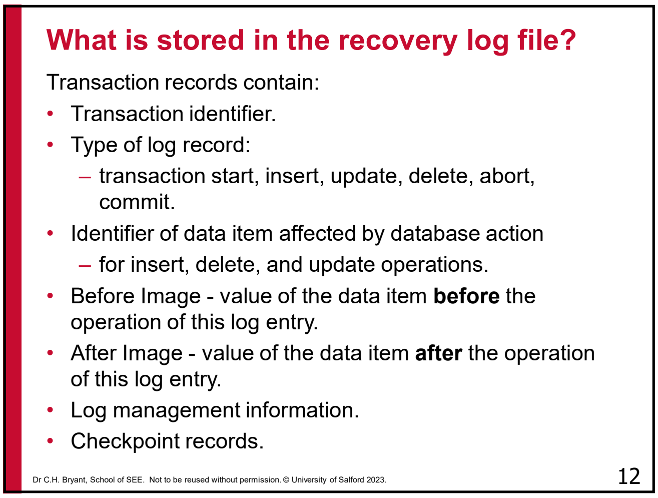
System Failure

- A transaction can commit once its writes are made to the database buffers.
- Updates made to the buffer are not automatically written to secondary storage (even for committed transactions).
- There may be a delay between committing and actual writing to secondary storage.
  - If system fails during this delay, the recovery manager must ensure that these updates reach the copy of the database on secondary storage.
- If a system failure occurs then the:
  - database buffers are lost;
  - the copy of the database on secondary storage survives, but it may be incorrect.



Recovery Log File (or Journal)

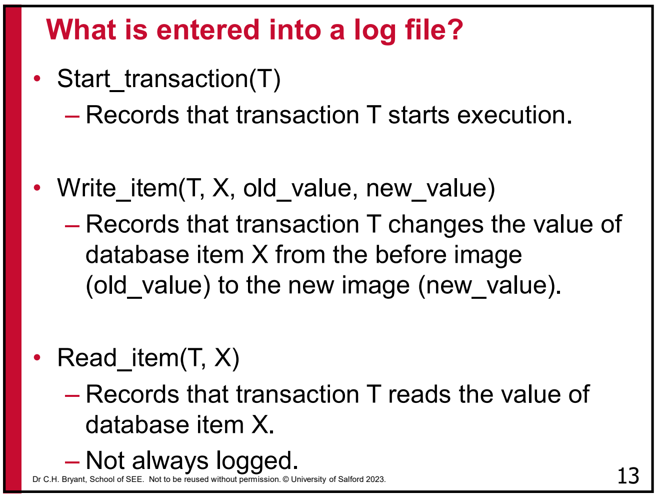
- To keep track of database transactions, the DBMS maintains a specific file called a log (or journal).
- Two or three copies of the log file are kept on secondary storage due to its importance in the recovery process.
- If the system fails, the log file is examined to see which transactions to **redo** and/or which transactions to **undo** or **ignore.**
- Several different protocols are used.
  - We will study these in the next lecture.



What is stored in the recovery log file?

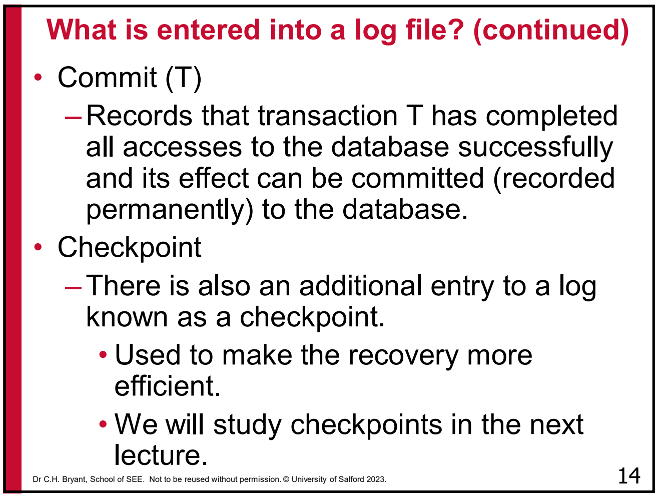
Transaction records contain:

- Transaction identifier.
- Type of log record:
  - transaction start, insert, update, delete, abort, commit.
- Identifier of data item affected by database action
  - for insert, delete, and update operations.
- Before Image - value of the data item **before** the operation of this log entry.
- After Image - value of the data item **after** the operation of this log entry.
- Log management information.
- Checkpoint records.



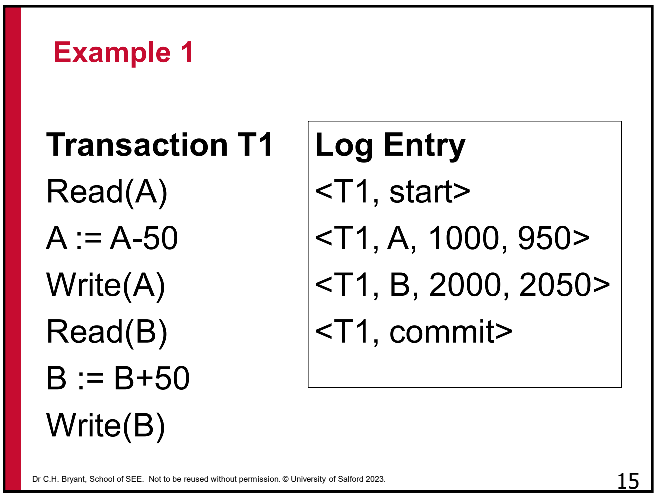
What is entered into a log file?

- Start_transaction(T)
  - Records that transaction T starts execution.
- Write_item(T, X, old_value, new_value)
  - Records that transaction T changes the value of database item X from the before image (old_value) to the new image (new_value).
- Read_item(T, X)
  - Records that transaction T reads the value of database item X.
  - Not always logged.



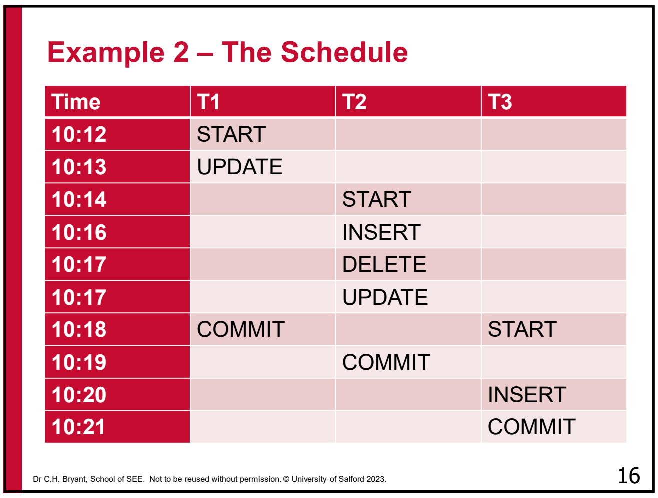
What is entered into a log file? (continued)

- Commit (T)
  - Records that transaction T has completed all accesses to the database successfully and its effect can be committed (recorded permanently) to the database.
- Checkpoint
  - There is also an additional entry to a log known as a checkpoint.
    - Used to make the recovery more efficient.
    - We will study checkpoints in the next lecture.



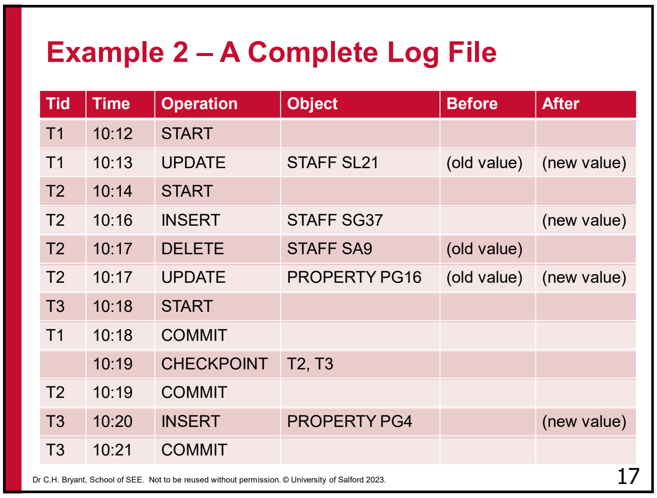
Example 1

Transaction T1	Log Entry
Read(A)	<T1, start>
A := A-50	<T1, A, 1000, 950>
Write(A)	<T1, B, 2000, 2050>
Read(B)	<T1, commit>
B := B+50	
Write(B)	



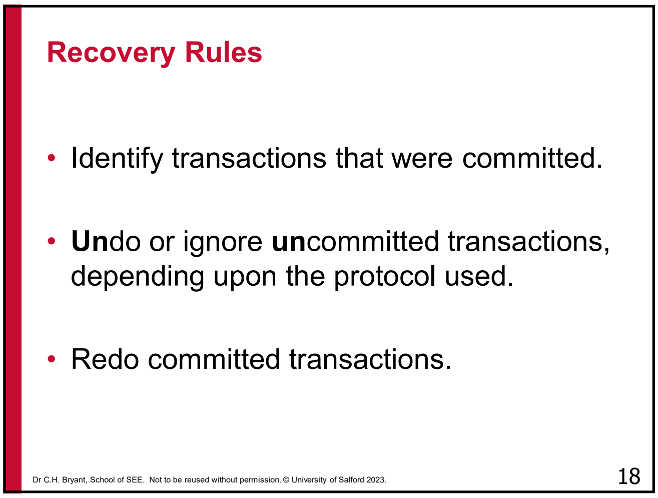
Example 2 – The Schedule

Time	T1	T2	T3
10:12	START		
10:13	UPDATE		
10:14		START	
10:16		INSERT	
10:17		DELETE	
10:17		UPDATE	
10:18	COMMIT		START
10:19		COMMIT	
10:20			INSERT
10:21			COMMIT



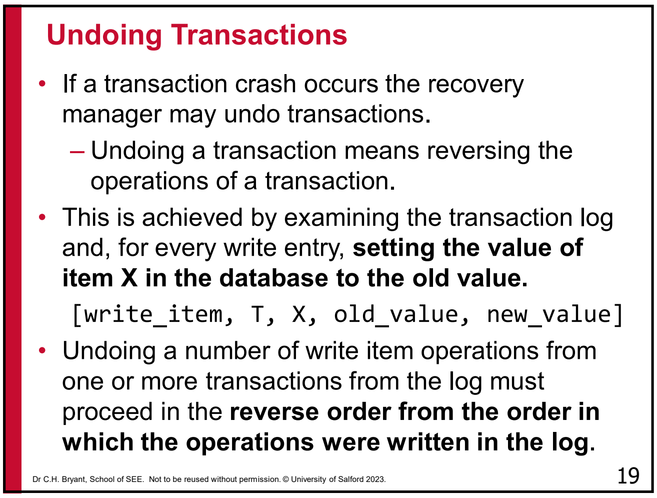
Example 2 – A Complete Log File

Tid	Time	Operation	Object	Before	After
T1	10:12	START			
T1	10:13	UPDATE	STAFF SL21	(old value)	(new value)
T2	10:14	START			
T2	10:16	INSERT	STAFF SG37		(new value)
T2	10:17	DELETE	STAFF SA9	(old value)	
T2	10:17	UPDATE	PROPERTY PG16	(old value)	(new value)
T3	10:18	START			
T1	10:18	COMMIT			
	10:19	CHECKPOINT	T2, T3		
T2	10:19	COMMIT			
T3	10:20	INSERT	PROPERTY PG4		(new value)
T3	10:21	COMMIT			



Recovery Rules

- Identify transactions that were committed.
- **Un**do or ignore **un**committed transactions, depending upon the protocol used.
- Redo committed transactions.



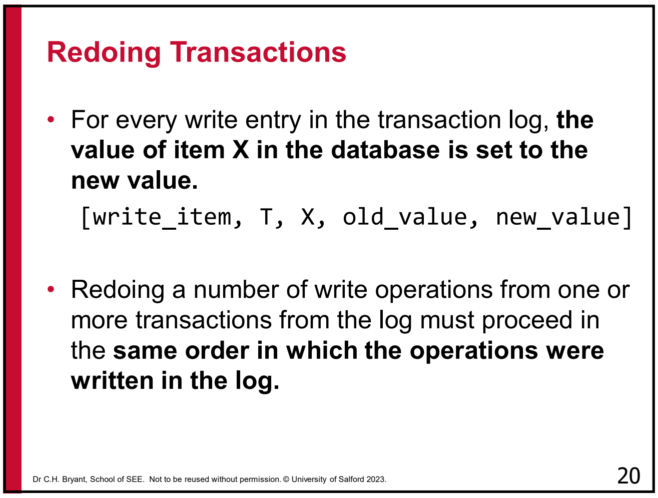
Undoing Transactions

- If a transaction crash occurs the recovery manager may undo transactions.
  - Undoing a transaction means reversing the operations of a transaction.
- This is achieved by examining the transaction log and, for every write entry, **setting the value of item X in the database to the old value.**

```
[write_item, T, X, old_value, new_value]
```

- Undoing a number of write item operations from one or more transactions from the log must proceed in the **reverse order from the order in which the operations were written in the log**.

19



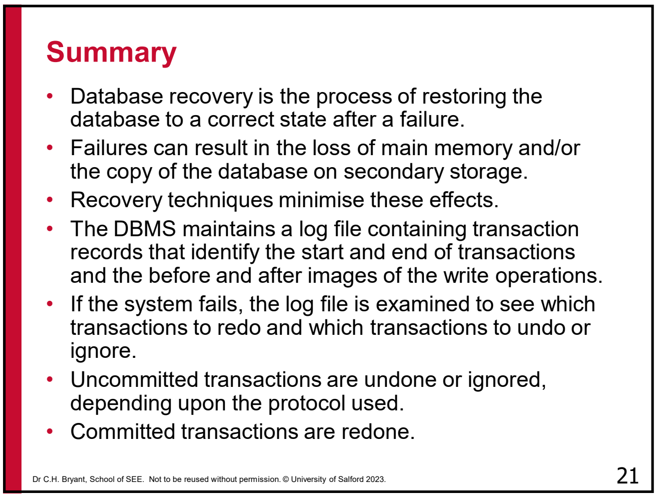
Redoing Transactions

- For every write entry in the transaction log, **the value of item X in the database is set to the new value.**

```
[write_item, T, X, old_value, new_value]
```

- Redoing a number of write operations from one or more transactions from the log must proceed in the **same order in which the operations were written in the log.**

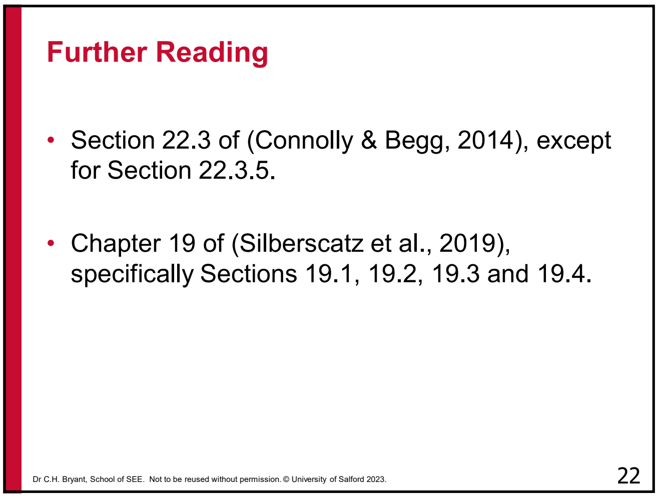
20



Summary

- Database recovery is the process of restoring the database to a correct state after a failure.
- Failures can result in the loss of main memory and/or the copy of the database on secondary storage.
- Recovery techniques minimise these effects.
- The DBMS maintains a log file containing transaction records that identify the start and end of transactions and the before and after images of the write operations.
- If the system fails, the log file is examined to see which transactions to redo and which transactions to undo or ignore.
- Uncommitted transactions are undone or ignored, depending upon the protocol used.
- Committed transactions are redone.

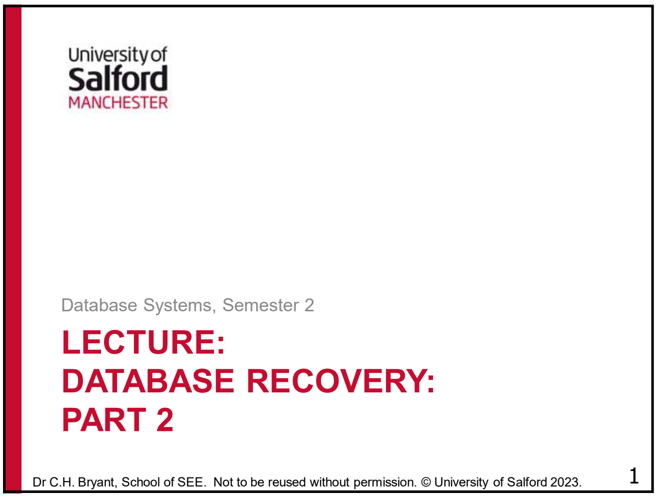
21



Further Reading

- Section 22.3 of (Connolly & Begg, 2014), except for Section 22.3.5.
- Chapter 19 of (Silberscatz et al., 2019), specifically Sections 19.1, 19.2, 19.3 and 19.4.

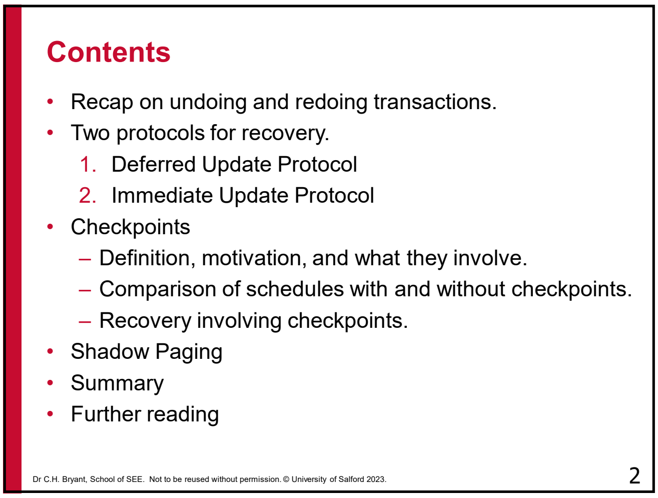
22



University of Salford MANCHESTER

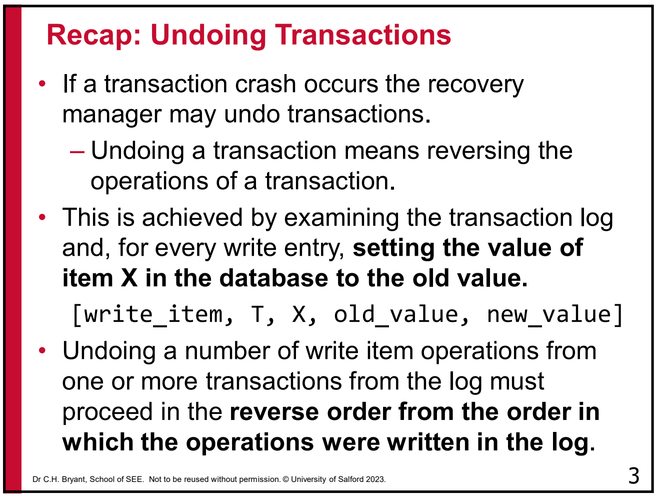
Database Systems, Semester 2

LECTURE: DATABASE RECOVERY: PART 2



Contents

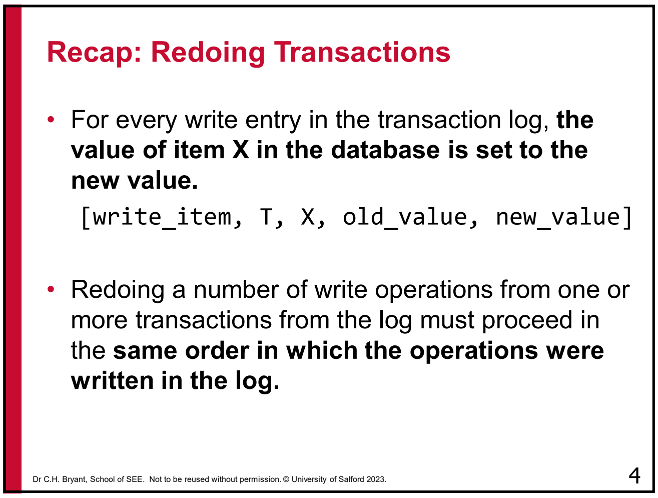
- Recap on undoing and redoing transactions.
- Two protocols for recovery.
  1. Deferred Update Protocol
  2. Immediate Update Protocol
- Checkpoints
  - Definition, motivation, and what they involve.
  - Comparison of schedules with and without checkpoints.
  - Recovery involving checkpoints.
- Shadow Paging
- Summary
- Further reading



Recap: Undoing Transactions

- If a transaction crash occurs the recovery manager may undo transactions.
  - Undoing a transaction means reversing the operations of a transaction.
- This is achieved by examining the transaction log and, for every write entry, **setting the value of item X in the database to the old value.**

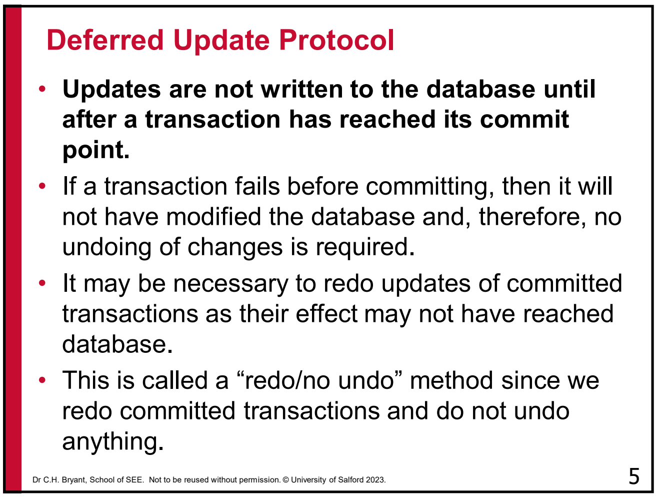
  `[write_item, T, X, old_value, new_value]`
- Undoing a number of write item operations from one or more transactions from the log must proceed in the **reverse order from the order in which the operations were written in the log**.



Recap: Redoing Transactions

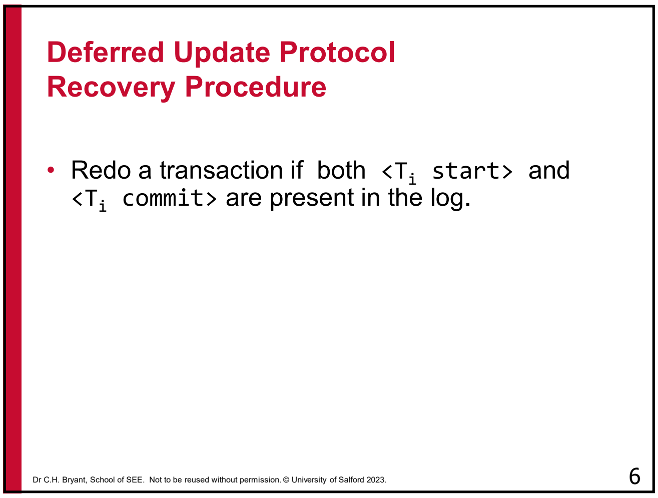
- For every write entry in the transaction log, **the value of item X in the database is set to the new value.**

  `[write_item, T, X, old_value, new_value]`
- Redoing a number of write operations from one or more transactions from the log must proceed in the **same order in which the operations were written in the log.**



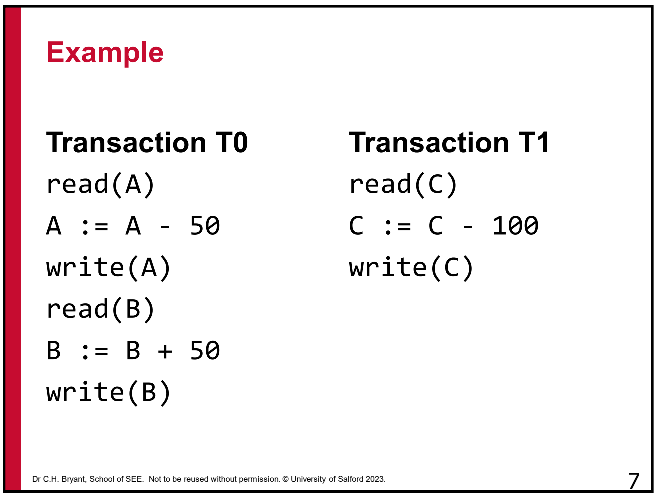
Deferred Update Protocol

- **Updates are not written to the database until after a transaction has reached its commit point.**
- If a transaction fails before committing, then it will not have modified the database and, therefore, no undoing of changes is required.
- It may be necessary to redo updates of committed transactions as their effect may not have reached database.
- This is called a “redo/no undo” method since we redo committed transactions and do not undo anything.



Deferred Update Protocol Recovery Procedure

- Redo a transaction if both <T_i start> and <T_i commit> are present in the log.

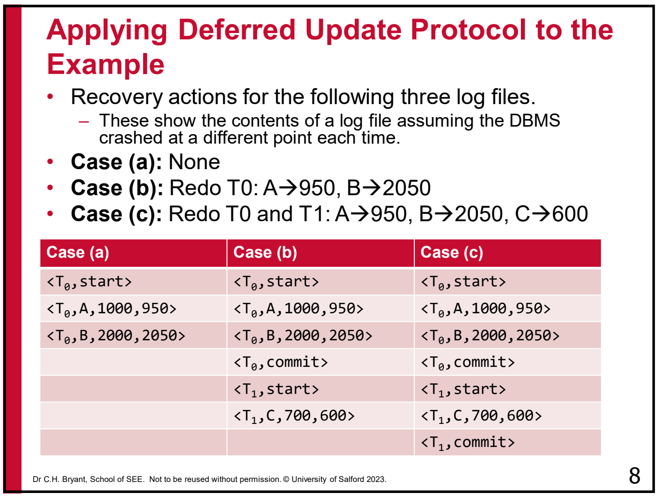


Example

Transaction T0	Transaction T1
read(A)	read(C)
A := A - 50	C := C - 100
write(A)	write(C)
read(B)	
B := B + 50	
write(B)	

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 7

7



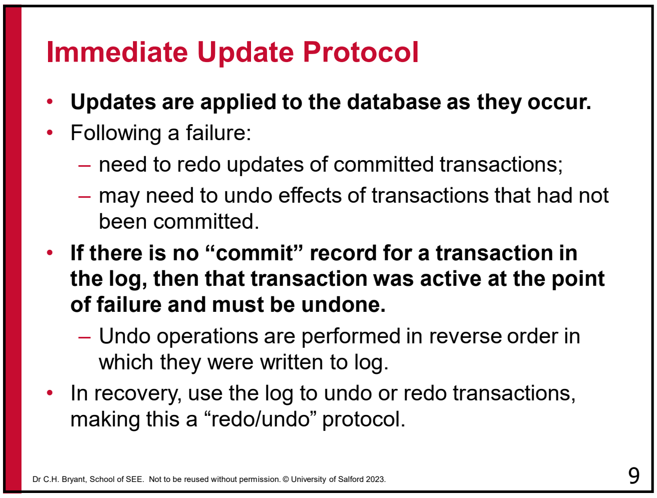
Applying Deferred Update Protocol to the Example

- Recovery actions for the following three log files.
  - These show the contents of a log file assuming the DBMS crashed at a different point each time.
- **Case (a):** None
- **Case (b):** Redo T0: A→950, B→2050
- **Case (c):** Redo T0 and T1: A→950, B→2050, C→600

Case (a)	Case (b)	Case (c)
<T_0,start>	<T_0,start>	<T_0,start>
<T_0,A,1000,950>	<T_0,A,1000,950>	<T_0,A,1000,950>
<T_0,B,2000,2050>	<T_0,B,2000,2050>	<T_0,B,2000,2050>
	<T_0,commit>	<T_0,commit>
	<T_1,start>	<T_1,start>
	<T_1,C,700,600>	<T_1,C,700,600>
		<T_1,commit>

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 8

8

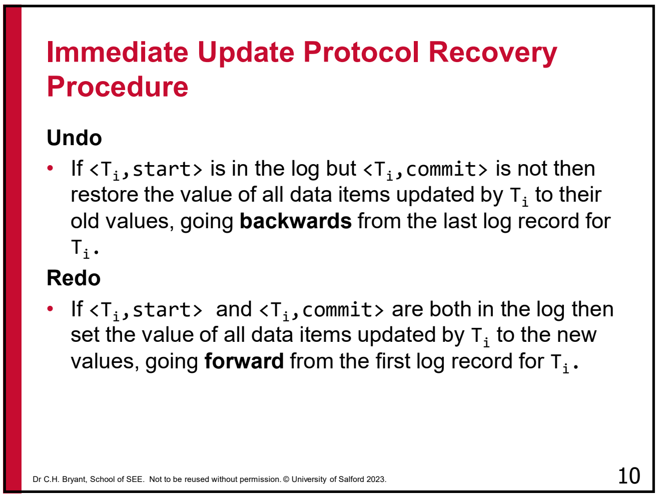


Immediate Update Protocol

- **Updates are applied to the database as they occur.**
- Following a failure:
  - need to redo updates of committed transactions;
  - may need to undo effects of transactions that had not been committed.
- **If there is no "commit" record for a transaction in the log, then that transaction was active at the point of failure and must be undone.**
  - Undo operations are performed in reverse order in which they were written to log.
- In recovery, use the log to undo or redo transactions, making this a "redo/undo" protocol.

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 9

9



Immediate Update Protocol Recovery Procedure

Undo

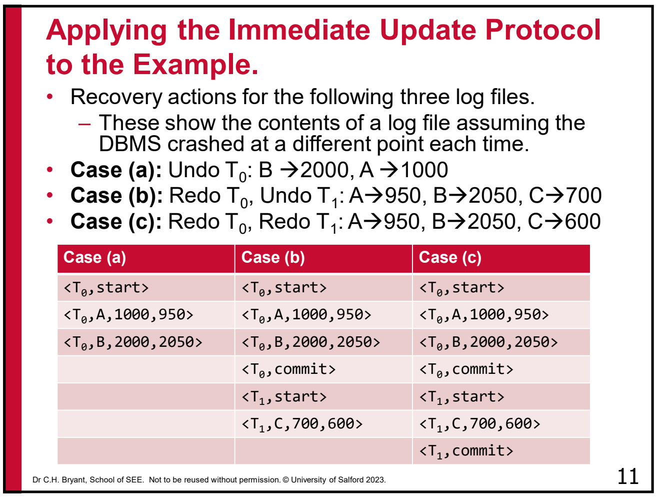
- If <T_i,start> is in the log but <T_i,commit> is not then restore the value of all data items updated by T_i to their old values, going **backwards** from the last log record for T_i.

Redo

- If <T_i,start> and <T_i,commit> are both in the log then set the value of all data items updated by T_i to the new values, going **forward** from the first log record for T_i.

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 10

10



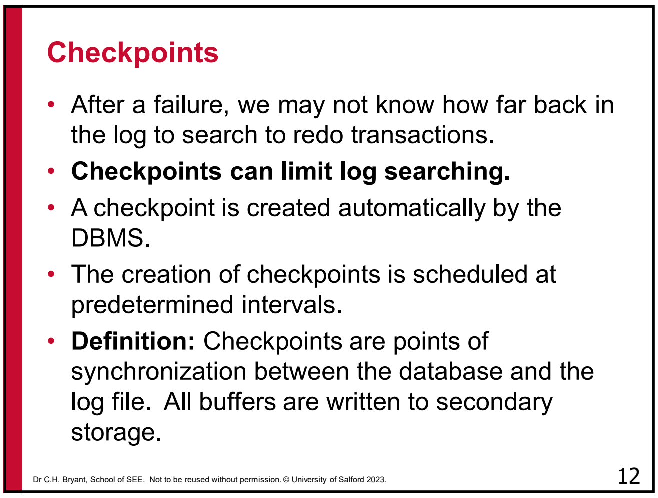
Applying the Immediate Update Protocol to the Example.

- Recovery actions for the following three log files.
  - These show the contents of a log file assuming the DBMS crashed at a different point each time.
- **Case (a):** Undo T_0: B →2000, A →1000
- **Case (b):** Redo T_0, Undo T_1: A→950, B→2050, C→700
- **Case (c):** Redo T_0, Redo T_1: A→950, B→2050, C→600

Case (a)	Case (b)	Case (c)
<T_0,start>	<T_0,start>	<T_0,start>
<T_0,A,1000,950>	<T_0,A,1000,950>	<T_0,A,1000,950>
<T_0,B,2000,2050>	<T_0,B,2000,2050>	<T_0,B,2000,2050>
	<T_0,commit>	<T_0,commit>
	<T_1,start>	<T_1,start>
	<T_1,C,700,600>	<T_1,C,700,600>
		<T_1,commit>

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 11

11

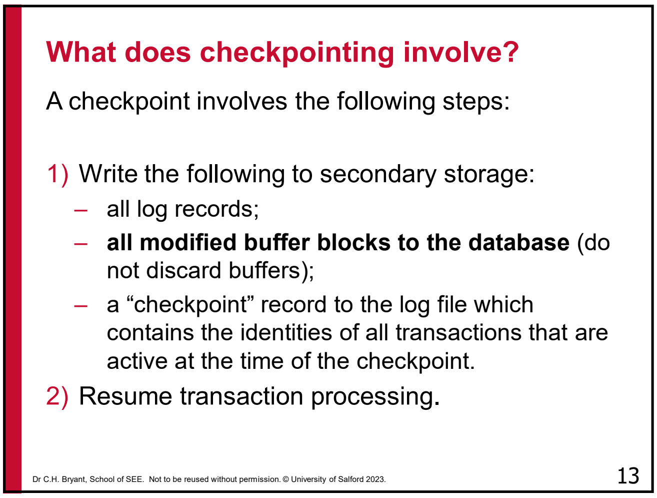


Checkpoints

- After a failure, we may not know how far back in the log to search to redo transactions.
- **Checkpoints can limit log searching.**
- A checkpoint is created automatically by the DBMS.
- The creation of checkpoints is scheduled at predetermined intervals.
- **Definition:** Checkpoints are points of synchronization between the database and the log file. All buffers are written to secondary storage.

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 12

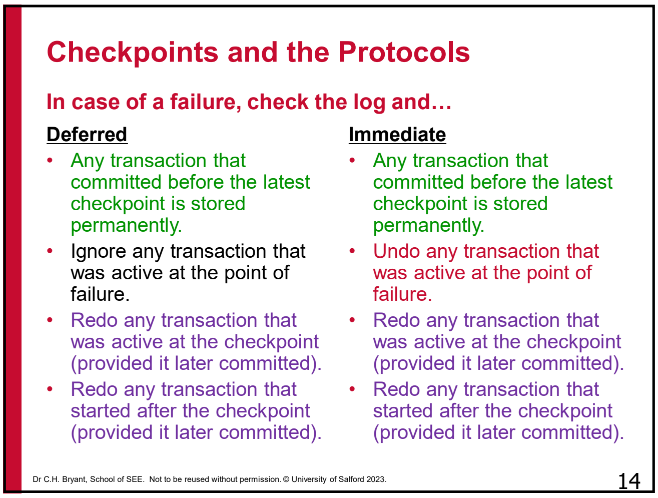
12



What does checkpointing involve?

A checkpoint involves the following steps:

1) Write the following to secondary storage:
   - all log records;
   - **all modified buffer blocks to the database** (do not discard buffers);
   - a "checkpoint" record to the log file which contains the identities of all transactions that are active at the time of the checkpoint.
2) Resume transaction processing.



Checkpoints and the Protocols

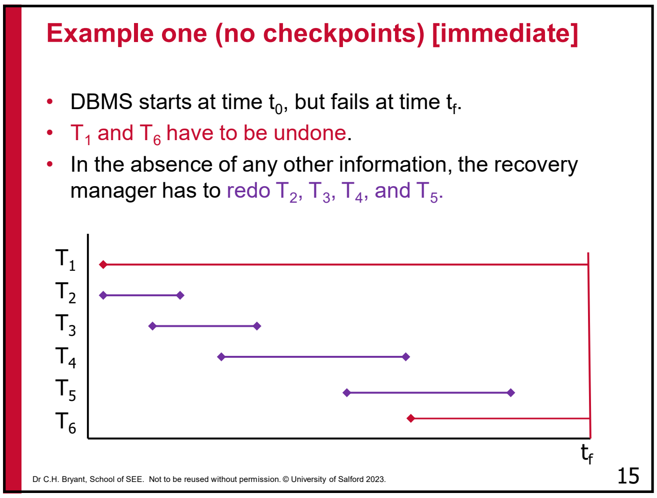
In case of a failure, check the log and…

Deferred

- Any transaction that committed before the latest checkpoint is stored permanently.
- Ignore any transaction that was active at the point of failure.
- Redo any transaction that was active at the checkpoint (provided it later committed).
- Redo any transaction that started after the checkpoint (provided it later committed).

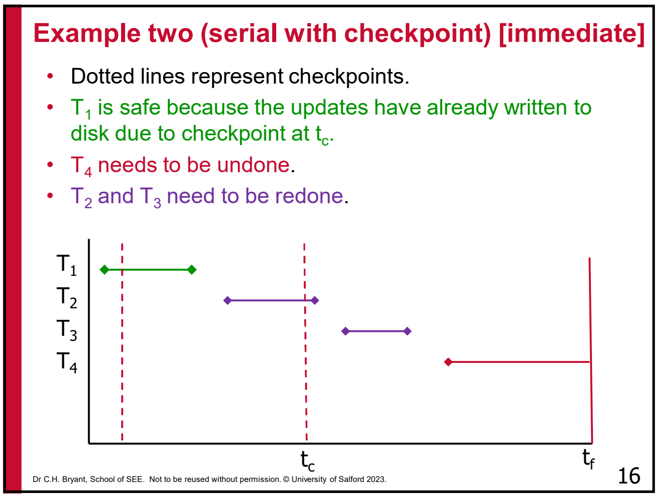
Immediate

- Any transaction that committed before the latest checkpoint is stored permanently.
- Undo any transaction that was active at the point of failure.
- Redo any transaction that was active at the checkpoint (provided it later committed).
- Redo any transaction that started after the checkpoint (provided it later committed).



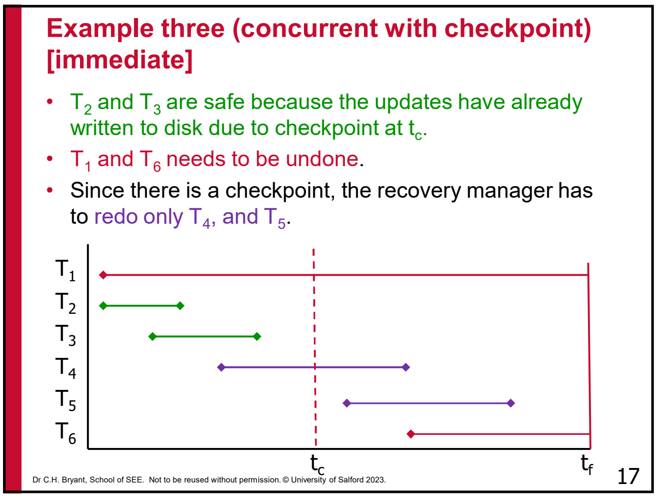
Example one (no checkpoints) [immediate]

- DBMS starts at time t_0, but fails at time t_f.
- T_1 and T_6 have to be undone.
- In the absence of any other information, the recovery manager has to redo T_2, T_3, T_4, and T_5.



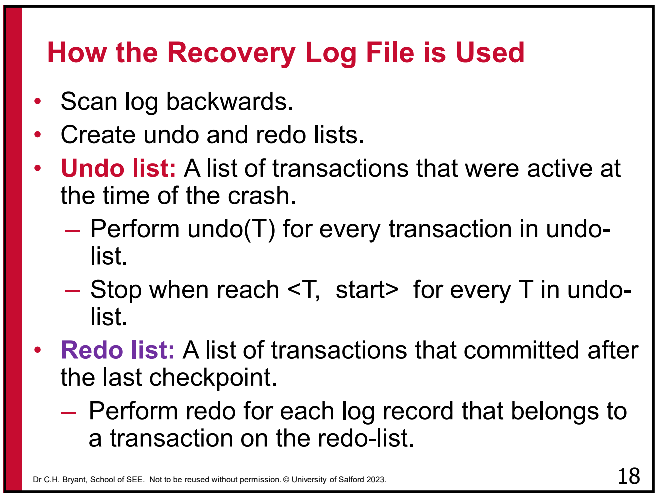
Example two (serial with checkpoint) [immediate]

- Dotted lines represent checkpoints.
- T_1 is safe because the updates have already written to disk due to checkpoint at t_c.
- T_4 needs to be undone.
- T_2 and T_3 need to be redone.



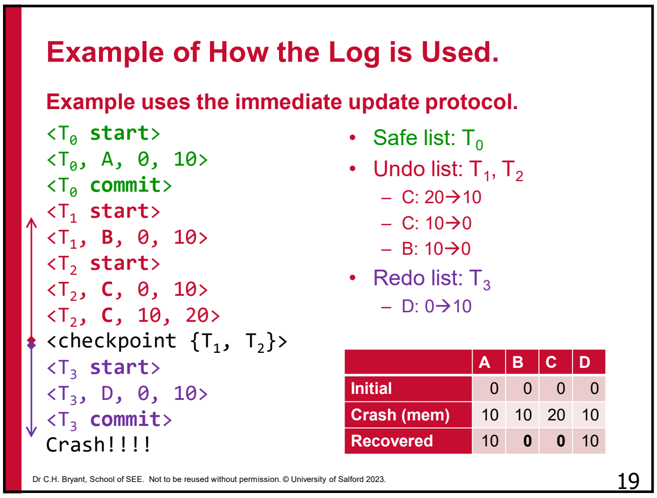
Example three (concurrent with checkpoint) [immediate]

- T_2 and T_3 are safe because the updates have already written to disk due to checkpoint at t_c.
- T_1 and T_6 needs to be undone.
- Since there is a checkpoint, the recovery manager has to redo only T_4, and T_5.



How the Recovery Log File is Used

- Scan log backwards.
- Create undo and redo lists.
- **Undo list:** A list of transactions that were active at the time of the crash.
  - Perform undo(T) for every transaction in undo-list.
  - Stop when reach <T, start> for every T in undo-list.
- **Redo list:** A list of transactions that committed after the last checkpoint.
  - Perform redo for each log record that belongs to a transaction on the redo-list.



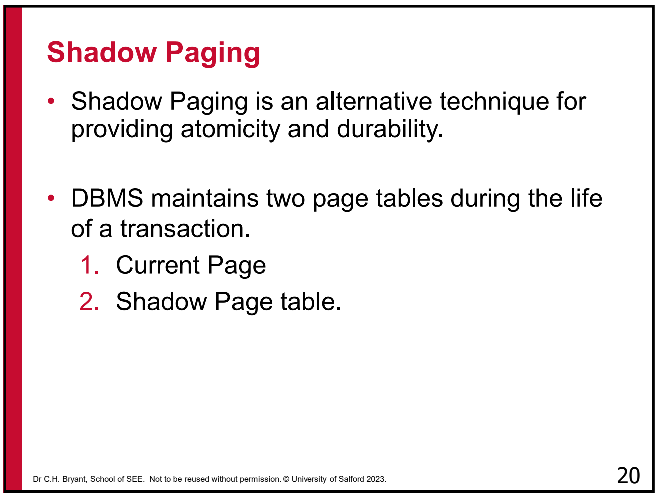
Example of How the Log is Used.

Example uses the immediate update protocol.

<T_0 **start**>
<T_0, A, 0, 10>
<T_0 **commit**>
<T_1 **start**>
<T_1, B, 0, 10>
<T_2 **start**>
<T_2, C, 0, 10>
<T_2, C, 10, 20>
<checkpoint {T_1, T_2}>
<T_3 **start**>
<T_3, D, 0, 10>
<T_3 **commit**>
Crash!!!!

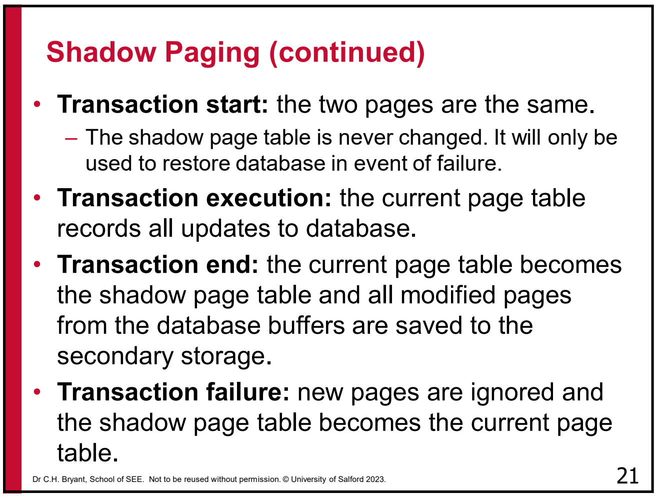
- Safe list: T_0
- Undo list: T_1, T_2
  - C: 20→10
  - C: 10→0
  - B: 10→0
- Redo list: T_3
  - D: 0→10

	A	B	C	D
Initial	0	0	0	0
Crash (mem)	10	10	20	10
Recovered	10	**0**	**0**	10



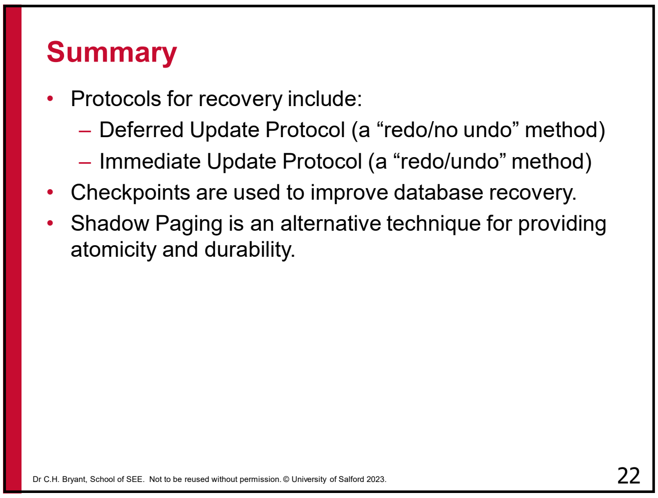
Shadow Paging

- Shadow Paging is an alternative technique for providing atomicity and durability.
- DBMS maintains two page tables during the life of a transaction.
  1. Current Page
  2. Shadow Page table.



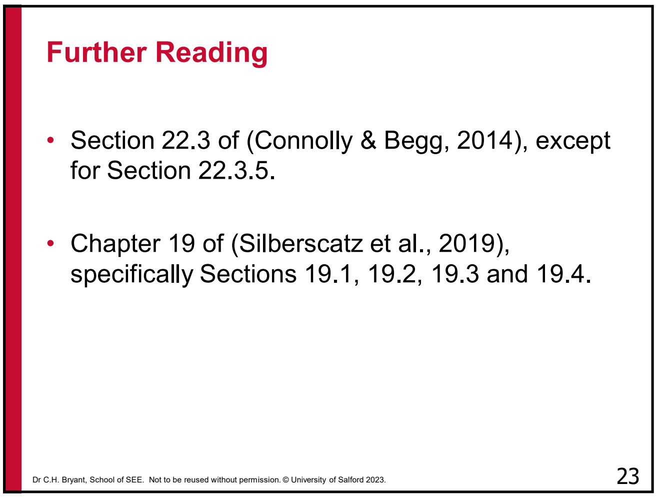
Shadow Paging (continued)

- **Transaction start:** the two pages are the same.
  - The shadow page table is never changed. It will only be used to restore database in event of failure.
- **Transaction execution:** the current page table records all updates to database.
- **Transaction end:** the current page table becomes the shadow page table and all modified pages from the database buffers are saved to the secondary storage.
- **Transaction failure:** new pages are ignored and the shadow page table becomes the current page table.



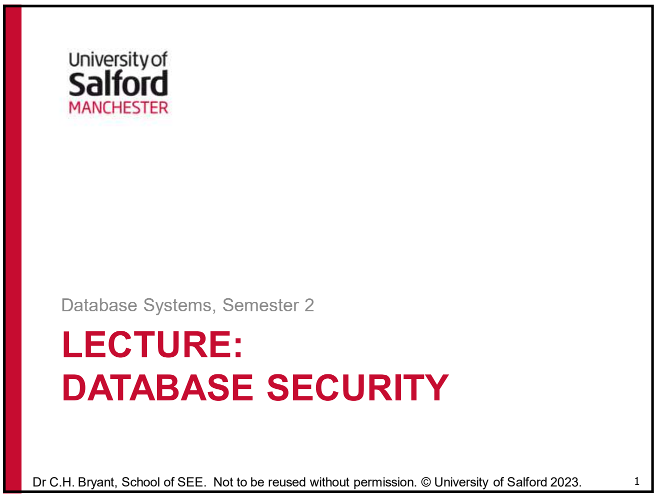
Summary

- Protocols for recovery include:
  - Deferred Update Protocol (a “redo/no undo” method)
  - Immediate Update Protocol (a “redo/undo” method)
- Checkpoints are used to improve database recovery.
- Shadow Paging is an alternative technique for providing atomicity and durability.



Further Reading

- Section 22.3 of (Connolly & Begg, 2014), except for Section 22.3.5.
- Chapter 19 of (Silberscatz et al., 2019), specifically Sections 19.1, 19.2, 19.3 and 19.4.

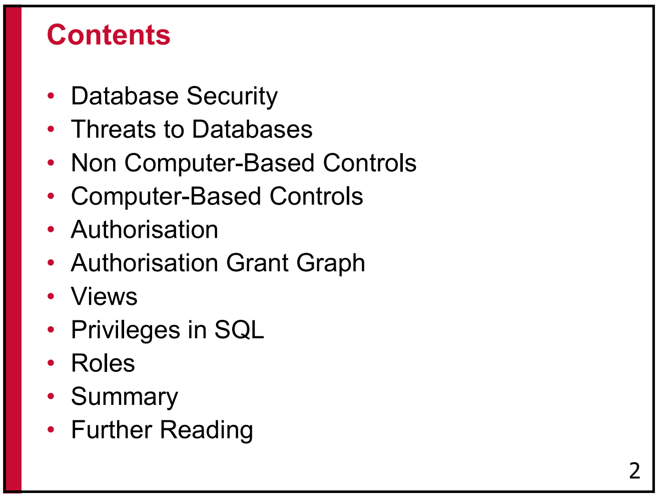


Database Systems, Semester 2

LECTURE: DATABASE SECURITY

Dr C.H. Bryant, School of SEE. Not to be reused without permission. © University of Salford 2023. 1

1

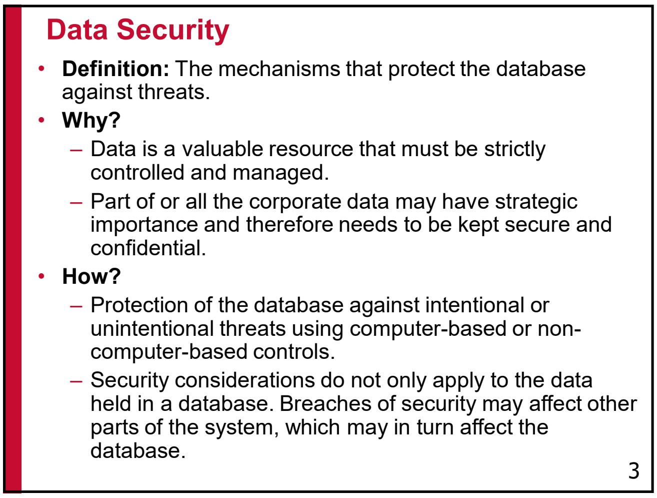


Contents

- Database Security
- Threats to Databases
- Non Computer-Based Controls
- Computer-Based Controls
- Authorisation
- Authorisation Grant Graph
- Views
- Privileges in SQL
- Roles
- Summary
- Further Reading

2

2

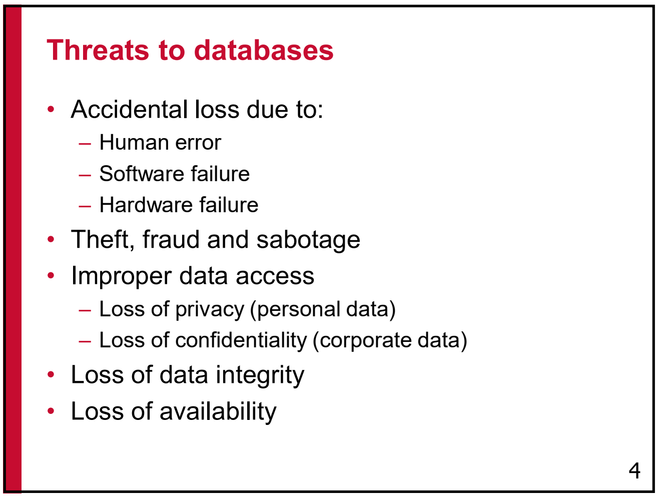


Data Security

- **Definition:** The mechanisms that protect the database against threats.
- **Why?**
  - Data is a valuable resource that must be strictly controlled and managed.
  - Part of or all the corporate data may have strategic importance and therefore needs to be kept secure and confidential.
- **How?**
  - Protection of the database against intentional or unintentional threats using computer-based or non-computer-based controls.
  - Security considerations do not only apply to the data held in a database. Breaches of security may affect other parts of the system, which may in turn affect the database.

3

3

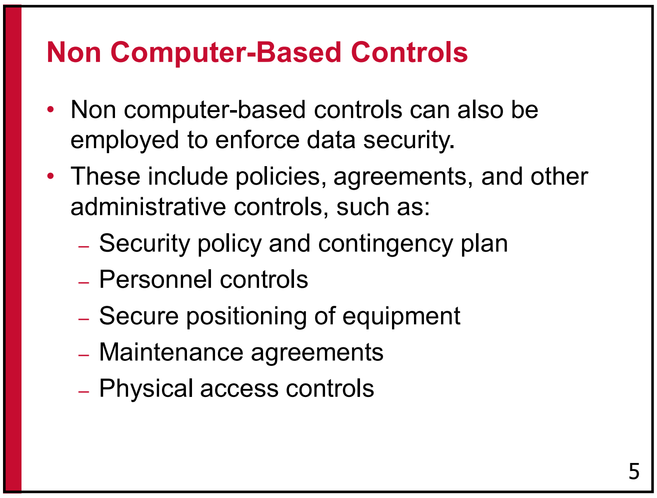


Threats to databases

- Accidental loss due to:
  - Human error
  - Software failure
  - Hardware failure
- Theft, fraud and sabotage
- Improper data access
  - Loss of privacy (personal data)
  - Loss of confidentiality (corporate data)
- Loss of data integrity
- Loss of availability

4

4

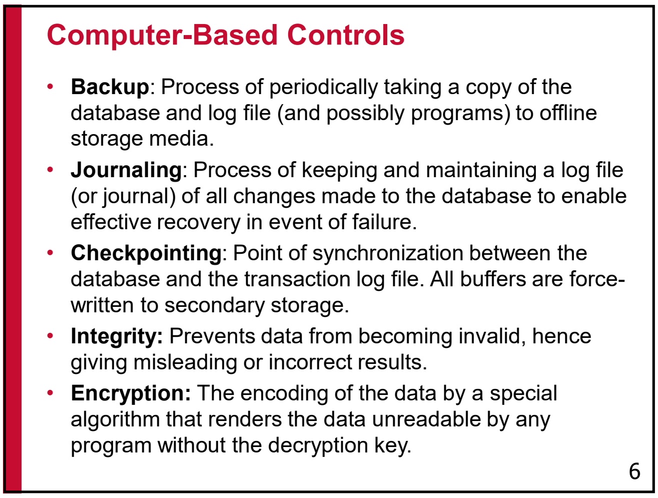


Non Computer-Based Controls

- Non computer-based controls can also be employed to enforce data security.
- These include policies, agreements, and other administrative controls, such as:
  - Security policy and contingency plan
  - Personnel controls
  - Secure positioning of equipment
  - Maintenance agreements
  - Physical access controls

5

5

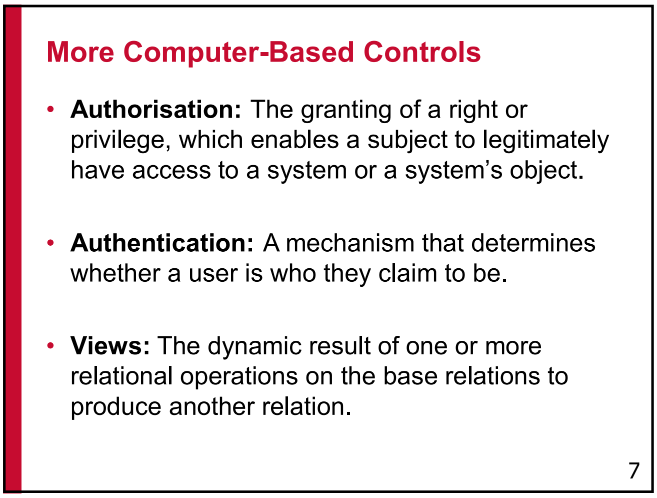


Computer-Based Controls

- **Backup**: Process of periodically taking a copy of the database and log file (and possibly programs) to offline storage media.
- **Journaling**: Process of keeping and maintaining a log file (or journal) of all changes made to the database to enable effective recovery in event of failure.
- **Checkpointing**: Point of synchronization between the database and the transaction log file. All buffers are force-written to secondary storage.
- **Integrity:** Prevents data from becoming invalid, hence giving misleading or incorrect results.
- **Encryption:** The encoding of the data by a special algorithm that renders the data unreadable by any program without the decryption key.

6

6

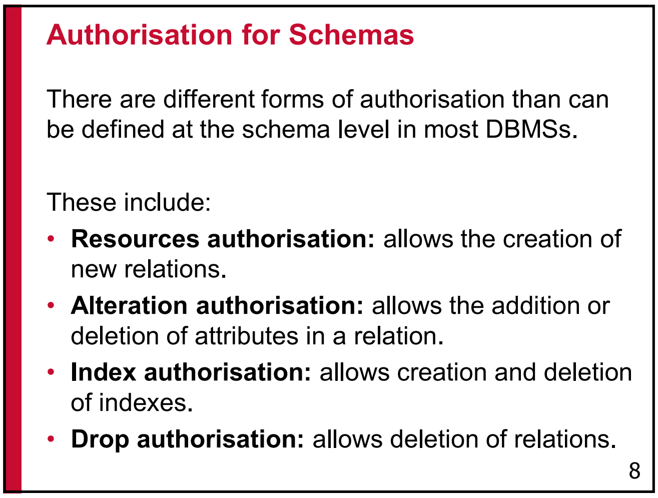


More Computer-Based Controls

- **Authorisation:** The granting of a right or privilege, which enables a subject to legitimately have access to a system or a system's object.
- **Authentication:** A mechanism that determines whether a user is who they claim to be.
- **Views:** The dynamic result of one or more relational operations on the base relations to produce another relation.

7

7



Authorisation for Schemas

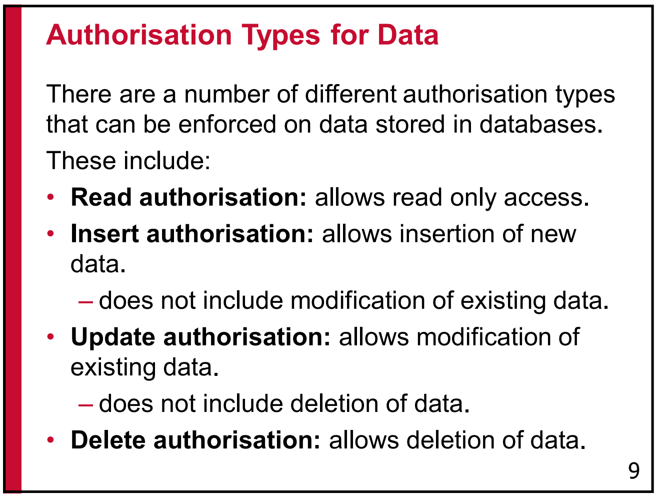
There are different forms of authorisation than can be defined at the schema level in most DBMSs.

These include:

- **Resources authorisation:** allows the creation of new relations.
- **Alteration authorisation:** allows the addition or deletion of attributes in a relation.
- **Index authorisation:** allows creation and deletion of indexes.
- **Drop authorisation:** allows deletion of relations.

8

8



Authorisation Types for Data

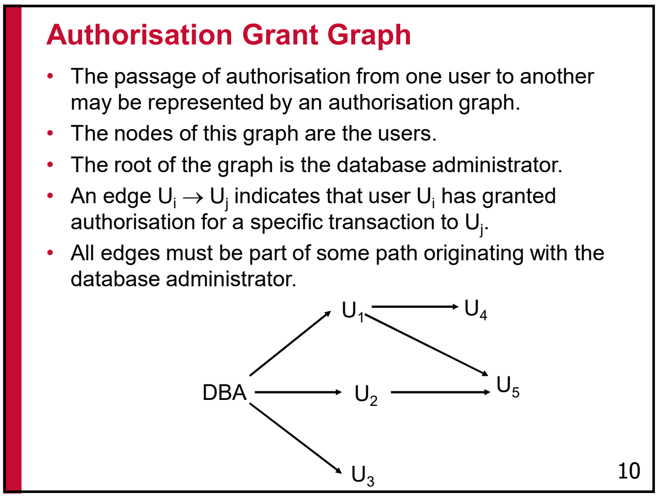
There are a number of different authorisation types that can be enforced on data stored in databases.

These include:

- **Read authorisation:** allows read only access.
- **Insert authorisation:** allows insertion of new data.
  - does not include modification of existing data.
- **Update authorisation:** allows modification of existing data.
  - does not include deletion of data.
- **Delete authorisation:** allows deletion of data.

9

9

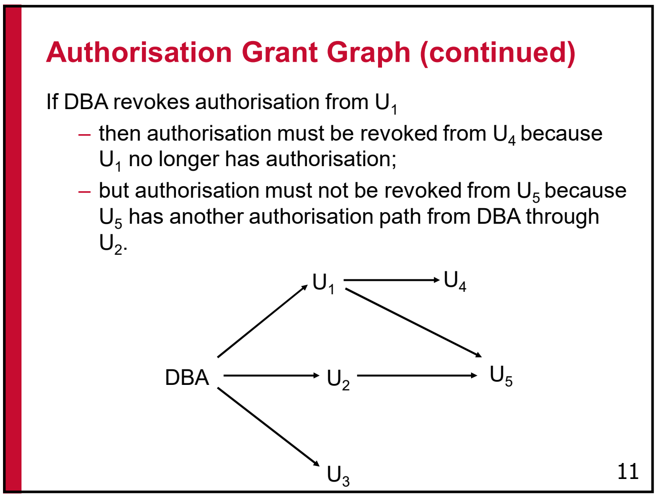


Authorisation Grant Graph

- The passage of authorisation from one user to another may be represented by an authorisation graph.
- The nodes of this graph are the users.
- The root of the graph is the database administrator.
- An edge $U_i \rightarrow U_j$ indicates that user U_i has granted authorisation for a specific transaction to U_j.
- All edges must be part of some path originating with the database administrator.

10

10



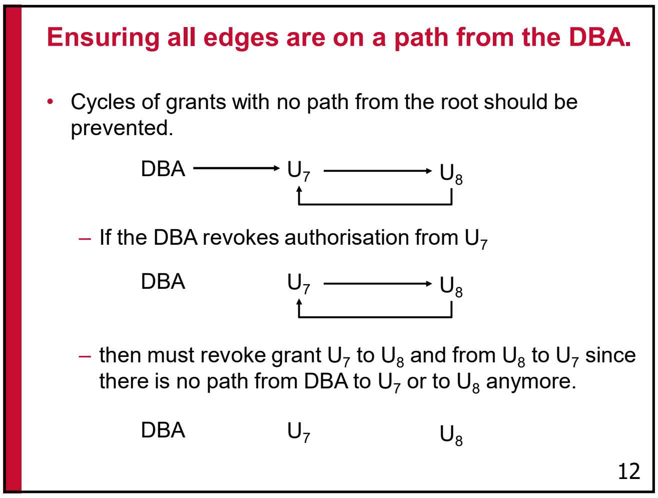
Authorisation Grant Graph (continued)

If DBA revokes authorisation from U_1

- then authorisation must be revoked from U_4 because U_1 no longer has authorisation;
- but authorisation must not be revoked from U_5 because U_5 has another authorisation path from DBA through U_2.

11

11



Ensuring all edges are on a path from the DBA.

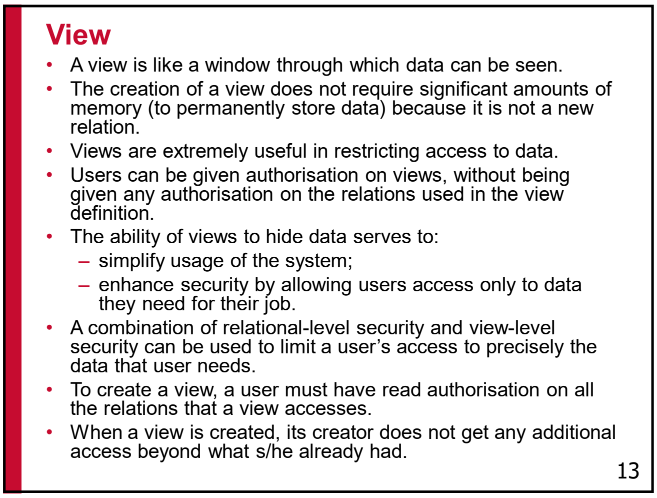
- Cycles of grants with no path from the root should be prevented.

- If the DBA revokes authorisation from U_7

- then must revoke grant U_7 to U_8 and from U_8 to U_7 since there is no path from DBA to U_7 or to U_8 anymore.

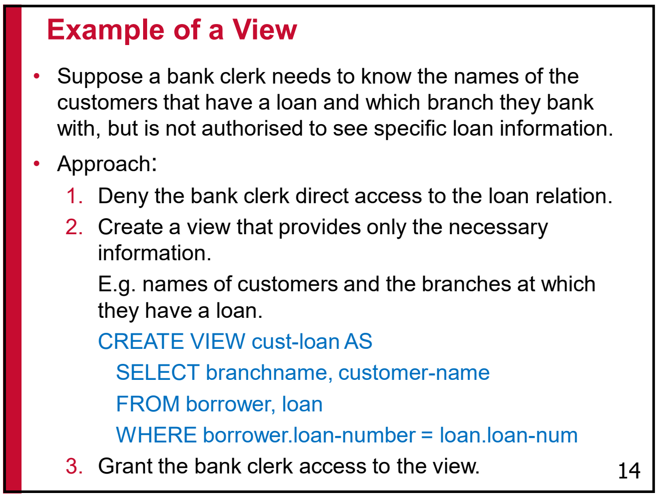
12

12



View

- A view is like a window through which data can be seen.
- The creation of a view does not require significant amounts of memory (to permanently store data) because it is not a new relation.
- Views are extremely useful in restricting access to data.
- Users can be given authorisation on views, without being given any authorisation on the relations used in the view definition.
- The ability of views to hide data serves to:
  - simplify usage of the system;
  - enhance security by allowing users access only to data they need for their job.
- A combination of relational-level security and view-level security can be used to limit a user's access to precisely the data that user needs.
- To create a view, a user must have read authorisation on all the relations that a view accesses.
- When a view is created, its creator does not get any additional access beyond what s/he already had.

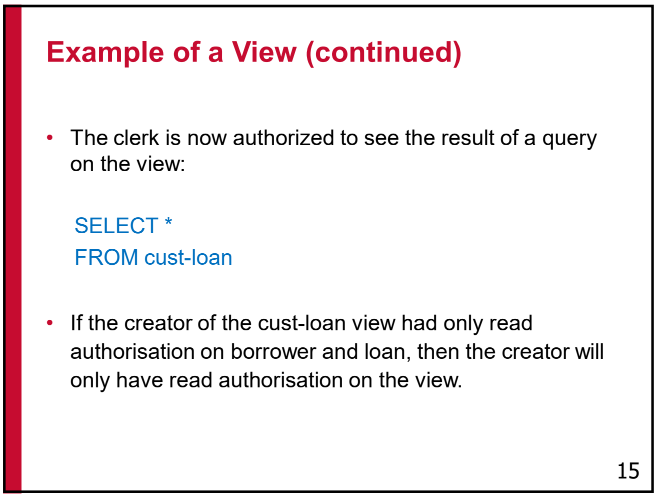


Example of a View

- Suppose a bank clerk needs to know the names of the customers that have a loan and which branch they bank with, but is not authorised to see specific loan information.
- Approach:
  1. Deny the bank clerk direct access to the loan relation.
  2. Create a view that provides only the necessary information.

     E.g. names of customers and the branches at which they have a loan.

     ```
     CREATE VIEW cust-loan AS
       SELECT branchname, customer-name
       FROM borrower, loan
       WHERE borrower.loan-number = loan.loan-num
     ```
  3. Grant the bank clerk access to the view.

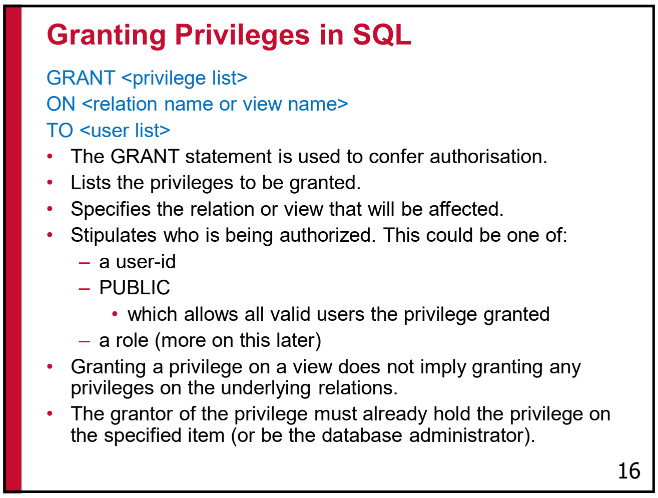


Example of a View (continued)

- The clerk is now authorized to see the result of a query on the view:

  ```
  SELECT *
  FROM cust-loan
  ```

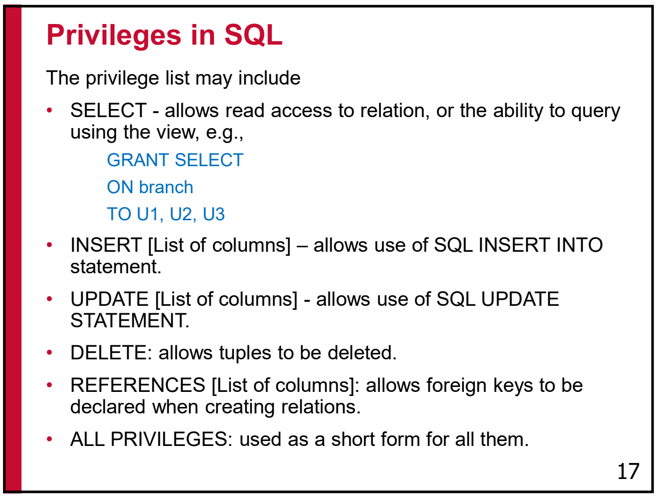
- If the creator of the cust-loan view had only read authorisation on borrower and loan, then the creator will only have read authorisation on the view.



Granting Privileges in SQL

```
GRANT <privilege list>
ON <relation name or view name>
TO <user list>
```

- The GRANT statement is used to confer authorisation.
- Lists the privileges to be granted.
- Specifies the relation or view that will be affected.
- Stipulates who is being authorized. This could be one of:
  - a user-id
  - PUBLIC
    - which allows all valid users the privilege granted
  - a role (more on this later)
- Granting a privilege on a view does not imply granting any privileges on the underlying relations.
- The grantor of the privilege must already hold the privilege on the specified item (or be the database administrator).

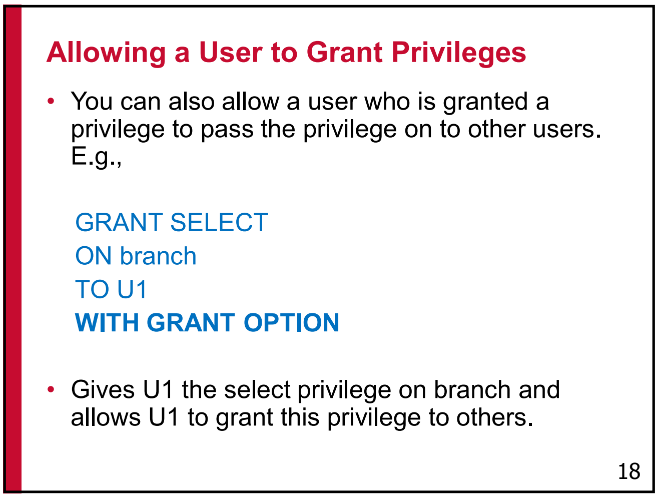


Privileges in SQL

The privilege list may include

- SELECT - allows read access to relation, or the ability to query using the view, e.g.,

  ```
  GRANT SELECT
  ON branch
  TO U1, U2, U3
  ```
- INSERT [List of columns] – allows use of SQL INSERT INTO statement.
- UPDATE [List of columns] - allows use of SQL UPDATE STATEMENT.
- DELETE: allows tuples to be deleted.
- REFERENCES [List of columns]: allows foreign keys to be declared when creating relations.
- ALL PRIVILEGES: used as a short form for all them.

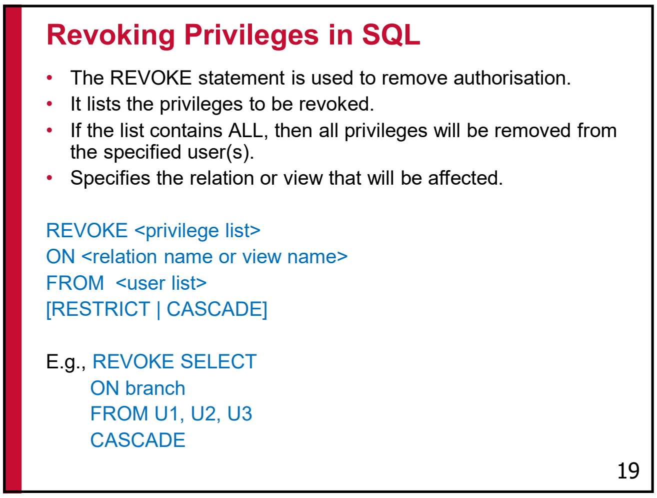


Allowing a User to Grant Privileges

- You can also allow a user who is granted a privilege to pass the privilege on to other users. E.g.,

  ```
  GRANT SELECT
  ON branch
  TO U1
  WITH GRANT OPTION
  ```

- Gives U1 the select privilege on branch and allows U1 to grant this privilege to others.

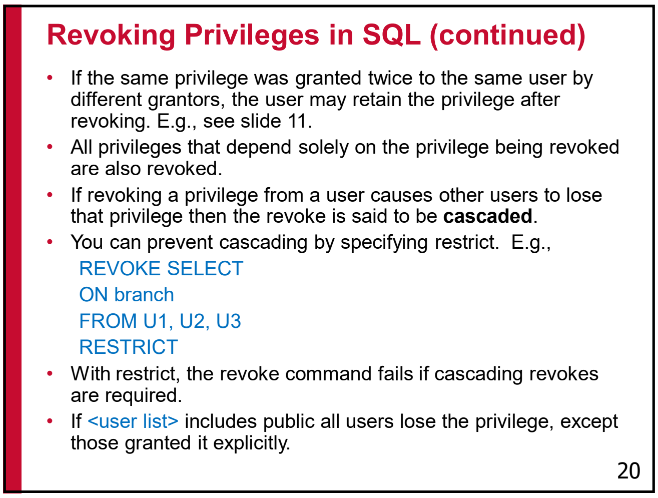


Revoking Privileges in SQL

- The REVOKE statement is used to remove authorisation.
- It lists the privileges to be revoked.
- If the list contains ALL, then all privileges will be removed from the specified user(s).
- Specifies the relation or view that will be affected.

```
REVOKE <privilege list>
ON <relation name or view name>
FROM  <user list>
[RESTRICT | CASCADE]
```

E.g.,
```
REVOKE SELECT
ON branch
FROM U1, U2, U3
CASCADE
```

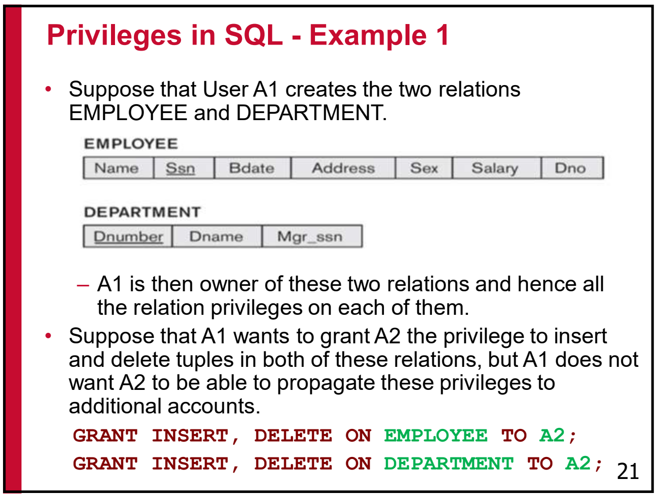


Revoking Privileges in SQL (continued)

- If the same privilege was granted twice to the same user by different grantors, the user may retain the privilege after revoking. E.g., see slide 11.
- All privileges that depend solely on the privilege being revoked are also revoked.
- If revoking a privilege from a user causes other users to lose that privilege then the revoke is said to be **cascaded**.
- You can prevent cascading by specifying restrict. E.g.,

```
REVOKE SELECT
ON branch
FROM U1, U2, U3
RESTRICT
```

- With restrict, the revoke command fails if cascading revokes are required.
- If <user list> includes public all users lose the privilege, except those granted it explicitly.



Privileges in SQL - Example 1

- Suppose that User A1 creates the two relations EMPLOYEE and DEPARTMENT.

EMPLOYEE

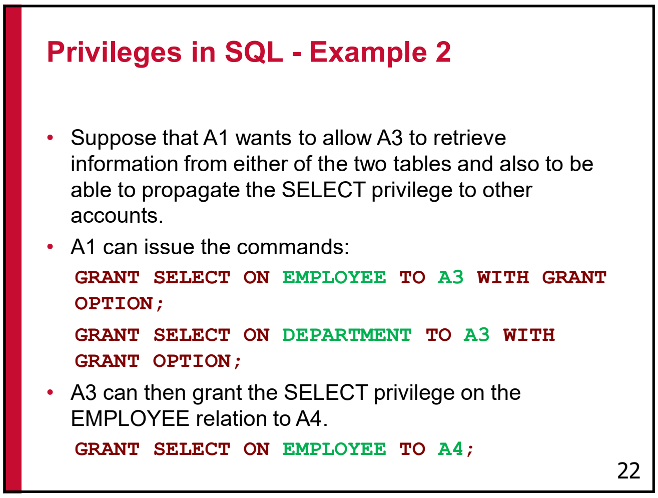
Name	Ssn	Bdate	Address	Sex	Salary	Dno

DEPARTMENT

Dnumber	Dname	Mgr_ssn

  - A1 is then owner of these two relations and hence all the relation privileges on each of them.
- Suppose that A1 wants to grant A2 the privilege to insert and delete tuples in both of these relations, but A1 does not want A2 to be able to propagate these privileges to additional accounts.

```
GRANT INSERT, DELETE ON EMPLOYEE TO A2;
GRANT INSERT, DELETE ON DEPARTMENT TO A2;
```



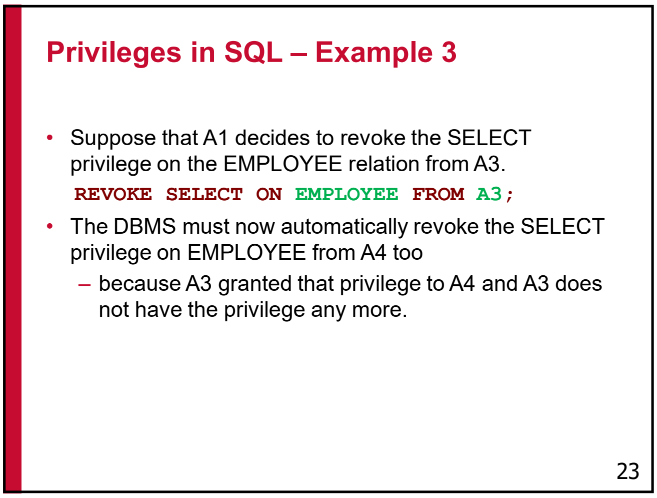
Privileges in SQL - Example 2

- Suppose that A1 wants to allow A3 to retrieve information from either of the two tables and also to be able to propagate the SELECT privilege to other accounts.
- A1 can issue the commands:

```
GRANT SELECT ON EMPLOYEE TO A3 WITH GRANT
OPTION;
GRANT SELECT ON DEPARTMENT TO A3 WITH
GRANT OPTION;
```

- A3 can then grant the SELECT privilege on the EMPLOYEE relation to A4.

```
GRANT SELECT ON EMPLOYEE TO A4;
```

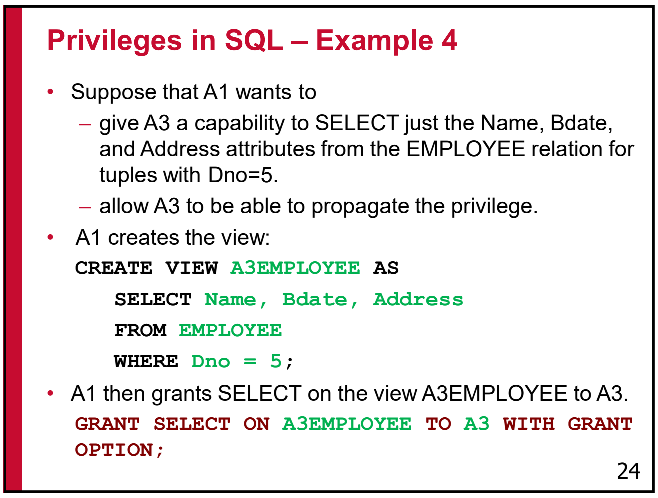


Privileges in SQL – Example 3

- Suppose that A1 decides to revoke the SELECT privilege on the EMPLOYEE relation from A3.

```
REVOKE SELECT ON EMPLOYEE FROM A3;
```

- The DBMS must now automatically revoke the SELECT privilege on EMPLOYEE from A4 too
  - because A3 granted that privilege to A4 and A3 does not have the privilege any more.



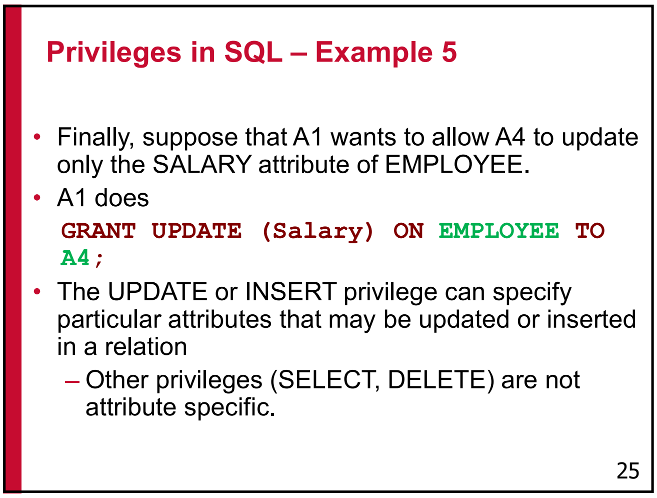
Privileges in SQL – Example 4

- Suppose that A1 wants to
  - give A3 a capability to SELECT just the Name, Bdate, and Address attributes from the EMPLOYEE relation for tuples with Dno=5.
  - allow A3 to be able to propagate the privilege.
- A1 creates the view:

```
CREATE VIEW A3EMPLOYEE AS
   SELECT Name, Bdate, Address
   FROM EMPLOYEE
   WHERE Dno = 5;
```

- A1 then grants SELECT on the view A3EMPLOYEE to A3.

```
GRANT SELECT ON A3EMPLOYEE TO A3 WITH GRANT
OPTION;
```

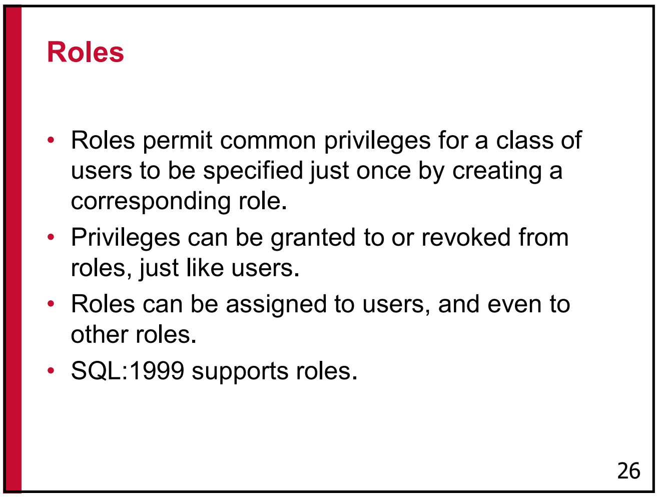


Privileges in SQL – Example 5

- Finally, suppose that A1 wants to allow A4 to update only the SALARY attribute of EMPLOYEE.
- A1 does

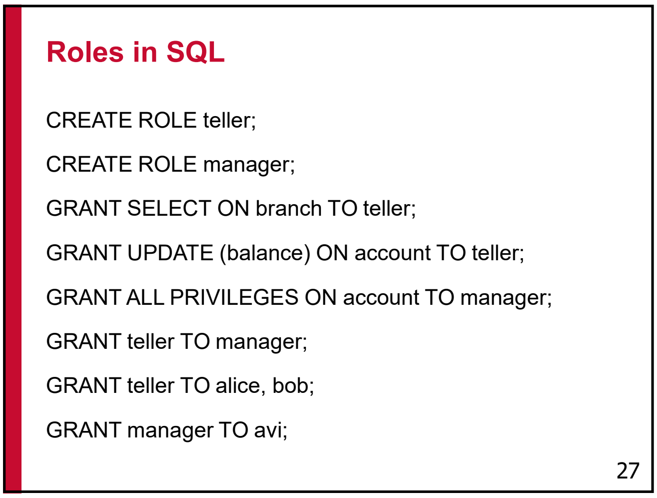
```
GRANT UPDATE (Salary) ON EMPLOYEE TO
A4;
```

- The UPDATE or INSERT privilege can specify particular attributes that may be updated or inserted in a relation
  - Other privileges (SELECT, DELETE) are not attribute specific.



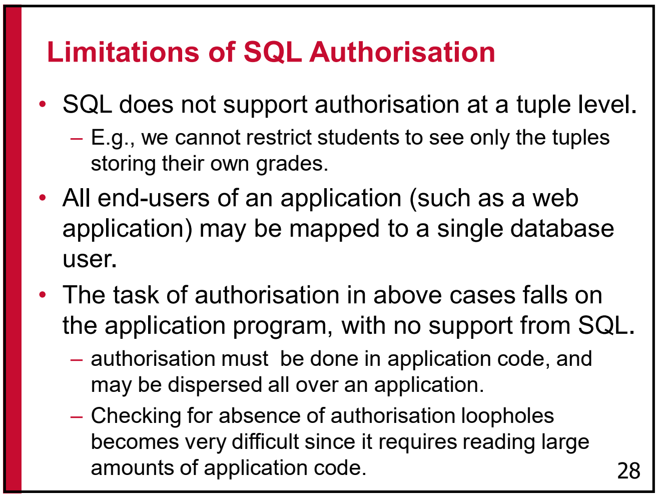
Roles

- Roles permit common privileges for a class of users to be specified just once by creating a corresponding role.
- Privileges can be granted to or revoked from roles, just like users.
- Roles can be assigned to users, and even to other roles.
- SQL:1999 supports roles.



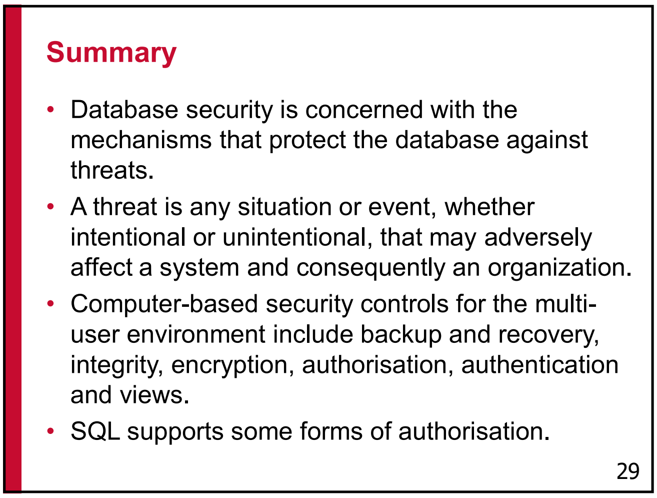
Roles in SQL

```
CREATE ROLE teller;
CREATE ROLE manager;
GRANT SELECT ON branch TO teller;
GRANT UPDATE (balance) ON account TO teller;
GRANT ALL PRIVILEGES ON account TO manager;
GRANT teller TO manager;
GRANT teller TO alice, bob;
GRANT manager TO avi;
```



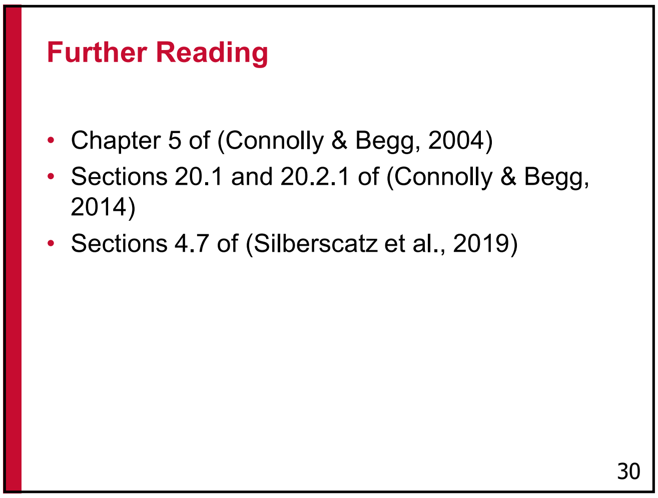
Limitations of SQL Authorisation

- SQL does not support authorisation at a tuple level.
  - E.g., we cannot restrict students to see only the tuples storing their own grades.
- All end-users of an application (such as a web application) may be mapped to a single database user.
- The task of authorisation in above cases falls on the application program, with no support from SQL.
  - authorisation must be done in application code, and may be dispersed all over an application.
  - Checking for absence of authorisation loopholes becomes very difficult since it requires reading large amounts of application code.



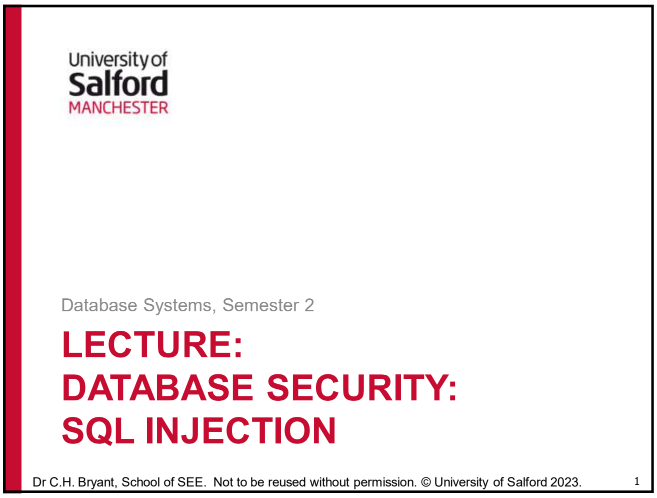
Summary

- Database security is concerned with the mechanisms that protect the database against threats.
- A threat is any situation or event, whether intentional or unintentional, that may adversely affect a system and consequently an organization.
- Computer-based security controls for the multi-user environment include backup and recovery, integrity, encryption, authorisation, authentication and views.
- SQL supports some forms of authorisation.



Further Reading

- Chapter 5 of (Connolly & Begg, 2004)
- Sections 20.1 and 20.2.1 of (Connolly & Begg, 2014)
- Sections 4.7 of (Silberscatz et al., 2019)

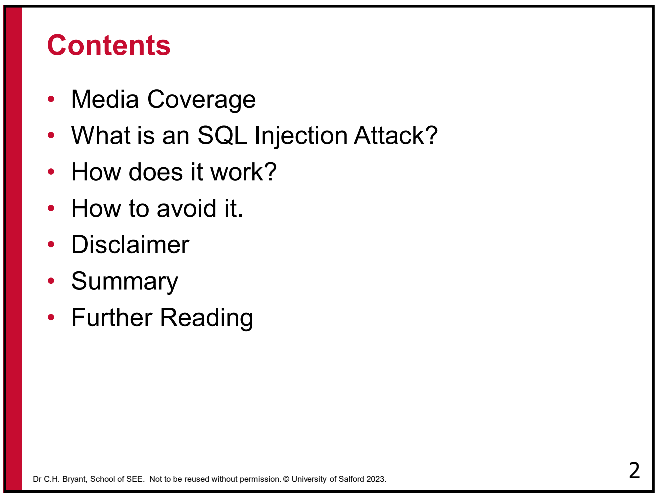


University of Salford MANCHESTER

Database Systems, Semester 2

LECTURE: DATABASE SECURITY: SQL INJECTION

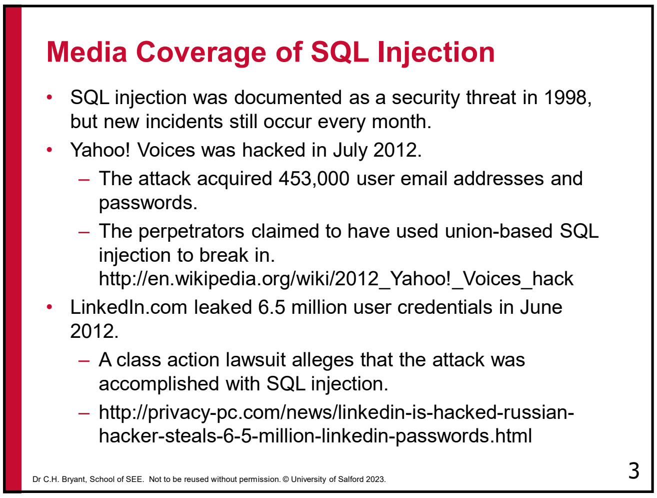
1



Contents

- Media Coverage
- What is an SQL Injection Attack?
- How does it work?
- How to avoid it.
- Disclaimer
- Summary
- Further Reading

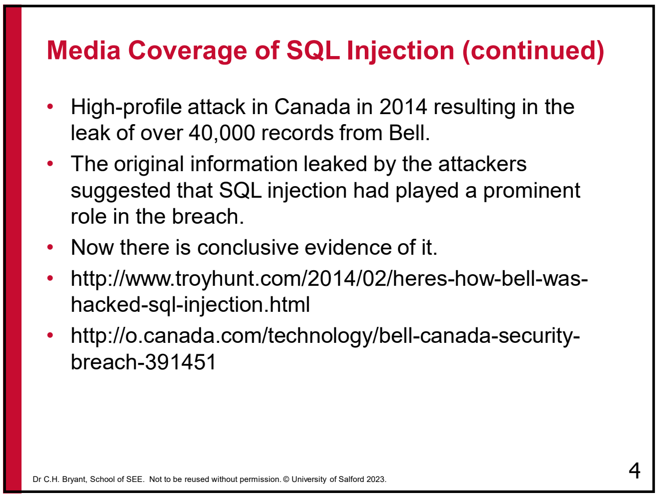
2



Media Coverage of SQL Injection

- SQL injection was documented as a security threat in 1998, but new incidents still occur every month.
- Yahoo! Voices was hacked in July 2012.
  - The attack acquired 453,000 user email addresses and passwords.
  - The perpetrators claimed to have used union-based SQL injection to break in. http://en.wikipedia.org/wiki/2012_Yahoo!_Voices_hack
- LinkedIn.com leaked 6.5 million user credentials in June 2012.
  - A class action lawsuit alleges that the attack was accomplished with SQL injection.
  - http://privacy-pc.com/news/linkedin-is-hacked-russian-hacker-steals-6-5-million-linkedin-passwords.html

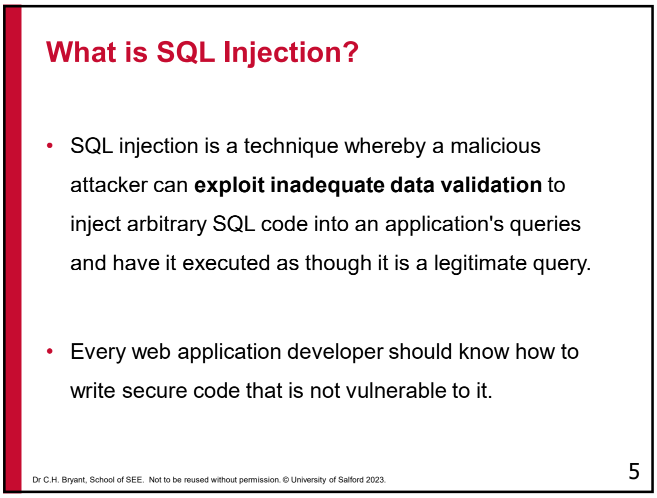
3



Media Coverage of SQL Injection (continued)

- High-profile attack in Canada in 2014 resulting in the leak of over 40,000 records from Bell.
- The original information leaked by the attackers suggested that SQL injection had played a prominent role in the breach.
- Now there is conclusive evidence of it.
- http://www.troyhunt.com/2014/02/heres-how-bell-was-hacked-sql-injection.html
- http://o.canada.com/technology/bell-canada-security-breach-391451

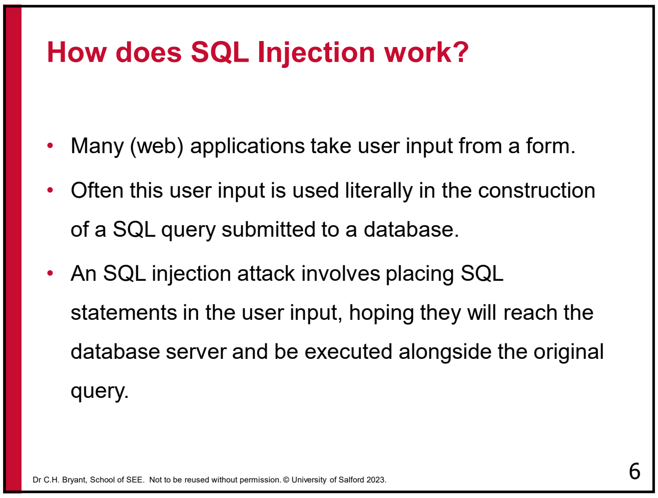
4



What is SQL Injection?

- SQL injection is a technique whereby a malicious attacker can **exploit inadequate data validation** to inject arbitrary SQL code into an application's queries and have it executed as though it is a legitimate query.
- Every web application developer should know how to write secure code that is not vulnerable to it.

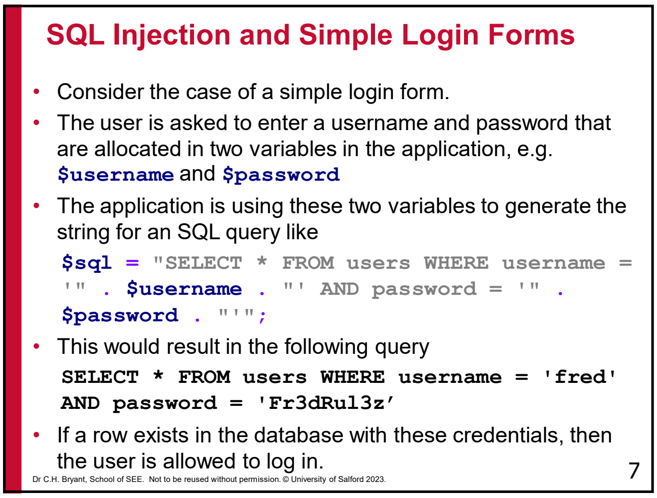
5



How does SQL Injection work?

- Many (web) applications take user input from a form.
- Often this user input is used literally in the construction of a SQL query submitted to a database.
- An SQL injection attack involves placing SQL statements in the user input, hoping they will reach the database server and be executed alongside the original query.

6



SQL Injection and Simple Login Forms

- Consider the case of a simple login form.
- The user is asked to enter a username and password that are allocated in two variables in the application, e.g. `$username` and `$password`
- The application is using these two variables to generate the string for an SQL query like

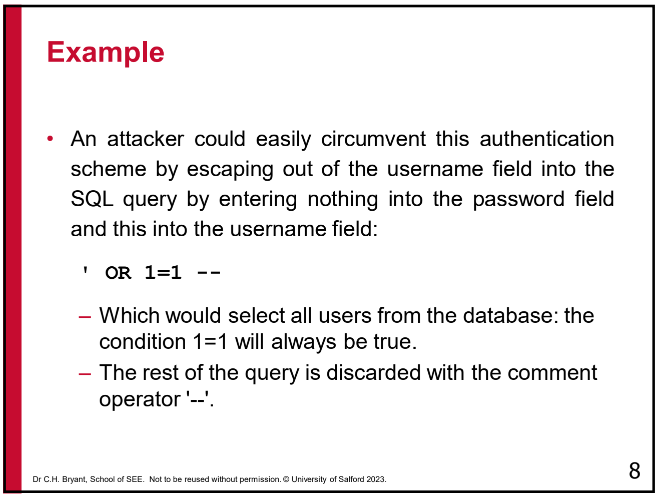
```
$sql = "SELECT * FROM users WHERE username =
'" . $username . "' AND password = '" .
$password . "'";
```

- This would result in the following query

```
SELECT * FROM users WHERE username = 'fred'
AND password = 'Fr3dRul3z'
```

- If a row exists in the database with these credentials, then the user is allowed to log in.

7



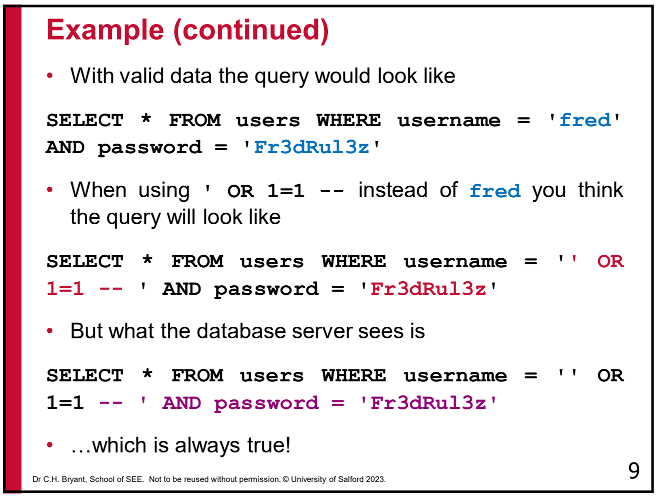
Example

- An attacker could easily circumvent this authentication scheme by escaping out of the username field into the SQL query by entering nothing into the password field and this into the username field:

```
' OR 1=1 --
```

  - Which would select all users from the database: the condition 1=1 will always be true.
  - The rest of the query is discarded with the comment operator '--'.

8



Example (continued)

- With valid data the query would look like

```
SELECT * FROM users WHERE username = 'fred'
AND password = 'Fr3dRul3z'
```

- When using `' OR 1=1 --` instead of `fred` you think the query will look like

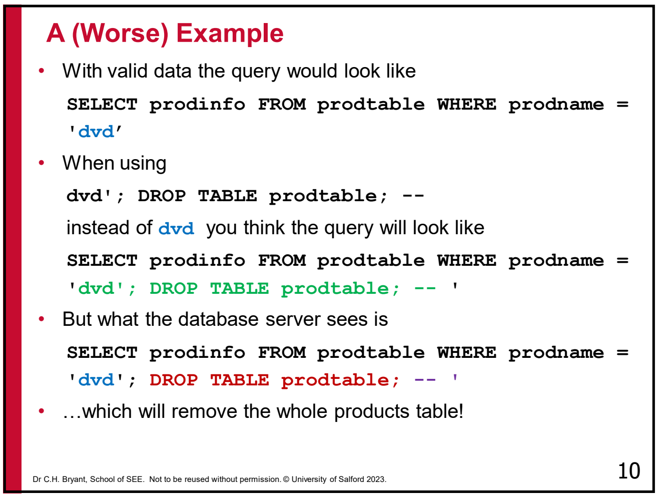
```
SELECT * FROM users WHERE username = '' OR
1=1 -- ' AND password = 'Fr3dRul3z'
```

- But what the database server sees is

```
SELECT * FROM users WHERE username = '' OR
1=1 -- ' AND password = 'Fr3dRul3z'
```

- …which is always true!

9



A (Worse) Example

- With valid data the query would look like

```
SELECT prodinfo FROM prodtable WHERE prodname =
'dvd'
```

- When using

```
dvd'; DROP TABLE prodtable; --
```

instead of `dvd` you think the query will look like

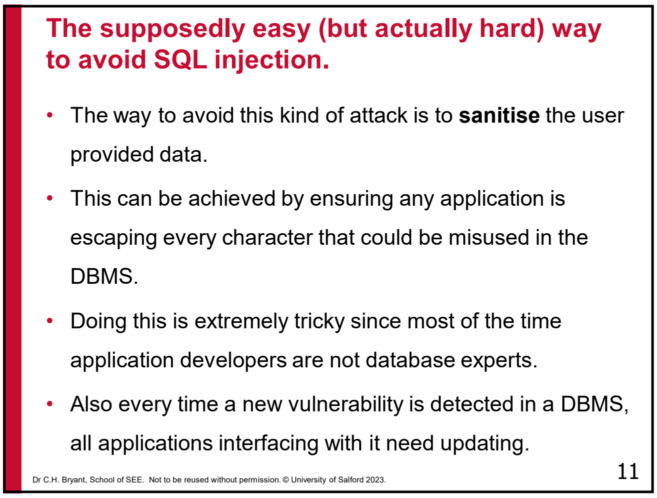
```
SELECT prodinfo FROM prodtable WHERE prodname =
'dvd'; DROP TABLE prodtable; -- '
```

- But what the database server sees is

```
SELECT prodinfo FROM prodtable WHERE prodname =
'dvd'; DROP TABLE prodtable; -- '
```

- …which will remove the whole products table!

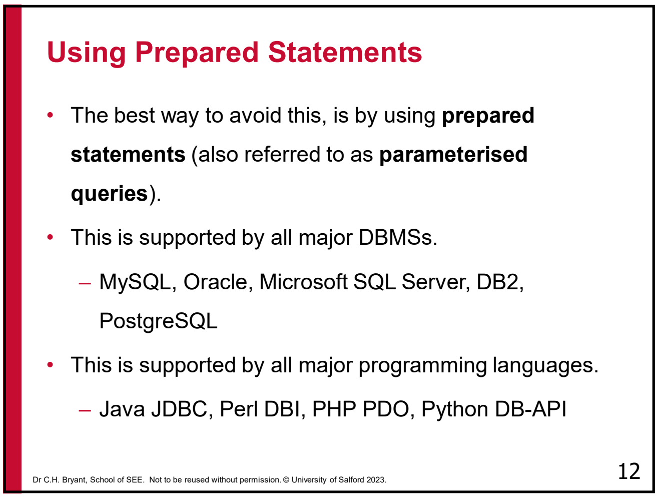
10



The supposedly easy (but actually hard) way to avoid SQL injection.

- The way to avoid this kind of attack is to **sanitise** the user provided data.
- This can be achieved by ensuring any application is escaping every character that could be misused in the DBMS.
- Doing this is extremely tricky since most of the time application developers are not database experts.
- Also every time a new vulnerability is detected in a DBMS, all applications interfacing with it need updating.

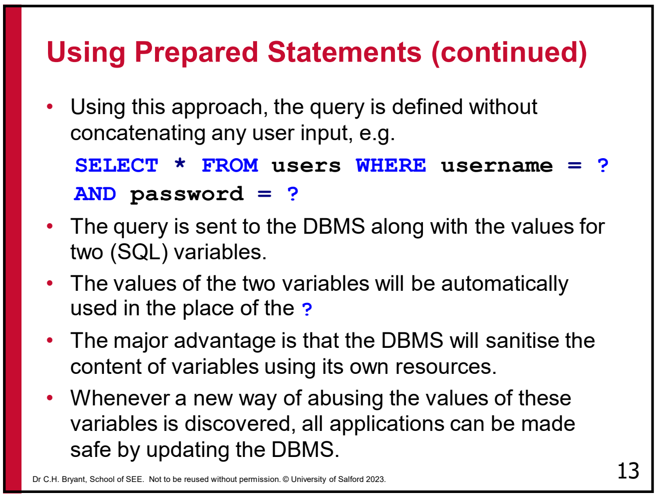
11



Using Prepared Statements

- The best way to avoid this, is by using **prepared statements** (also referred to as **parameterised queries**).
- This is supported by all major DBMSs.
  - MySQL, Oracle, Microsoft SQL Server, DB2, PostgreSQL
- This is supported by all major programming languages.
  - Java JDBC, Perl DBI, PHP PDO, Python DB-API

12



Using Prepared Statements (continued)

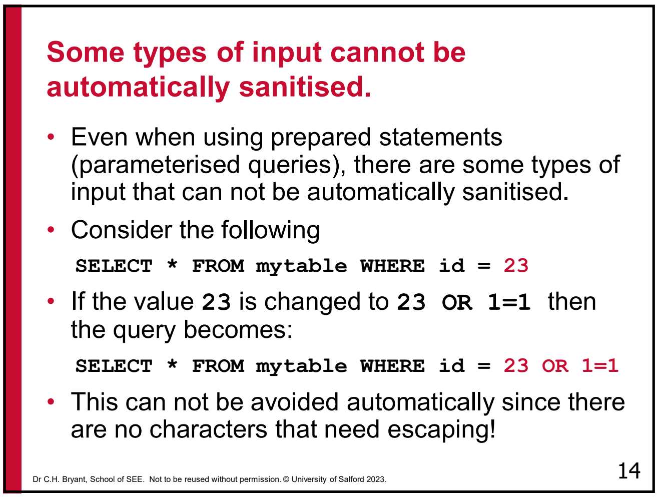
- Using this approach, the query is defined without concatenating any user input, e.g.

```
SELECT * FROM users WHERE username = ?
AND password = ?
```

- The query is sent to the DBMS along with the values for two (SQL) variables.
- The values of the two variables will be automatically used in the place of the ?
- The major advantage is that the DBMS will sanitise the content of variables using its own resources.
- Whenever a new way of abusing the values of these variables is discovered, all applications can be made safe by updating the DBMS.

13

13



Some types of input cannot be automatically sanitised.

- Even when using prepared statements (parameterised queries), there are some types of input that can not be automatically sanitised.
- Consider the following

```
SELECT * FROM mytable WHERE id = 23
```

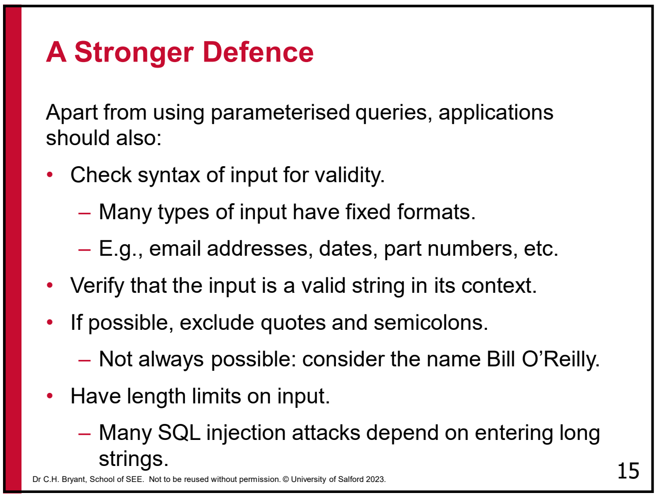
- If the value `23` is changed to `23 OR 1=1` then the query becomes:

```
SELECT * FROM mytable WHERE id = 23 OR 1=1
```

- This can not be avoided automatically since there are no characters that need escaping!

14

14



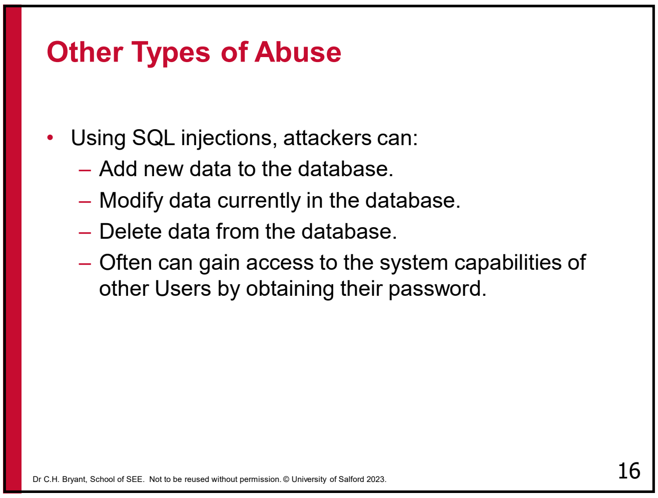
A Stronger Defence

Apart from using parameterised queries, applications should also:

- Check syntax of input for validity.
  - Many types of input have fixed formats.
  - E.g., email addresses, dates, part numbers, etc.
- Verify that the input is a valid string in its context.
- If possible, exclude quotes and semicolons.
  - Not always possible: consider the name Bill O'Reilly.
- Have length limits on input.
  - Many SQL injection attacks depend on entering long strings.

15

15

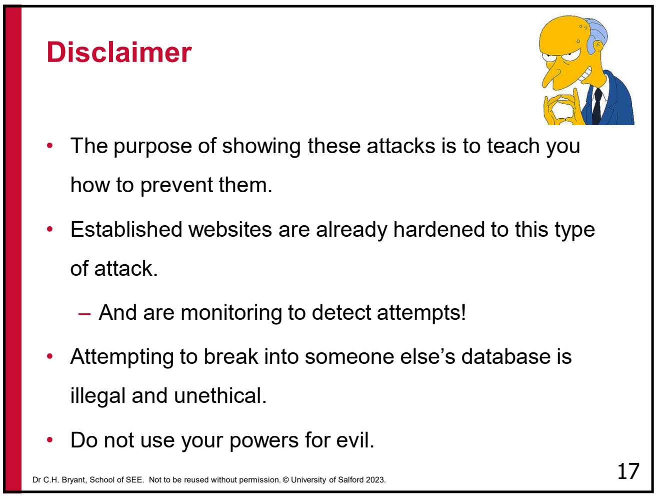


Other Types of Abuse

- Using SQL injections, attackers can:
  - Add new data to the database.
  - Modify data currently in the database.
  - Delete data from the database.
  - Often can gain access to the system capabilities of other Users by obtaining their password.

16

16

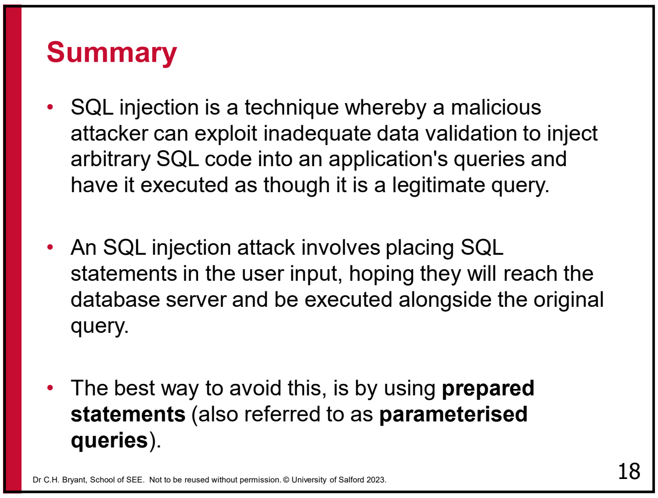


Disclaimer

- The purpose of showing these attacks is to teach you how to prevent them.
- Established websites are already hardened to this type of attack.
  - And are monitoring to detect attempts!
- Attempting to break into someone else's database is illegal and unethical.
- Do not use your powers for evil.

17

17

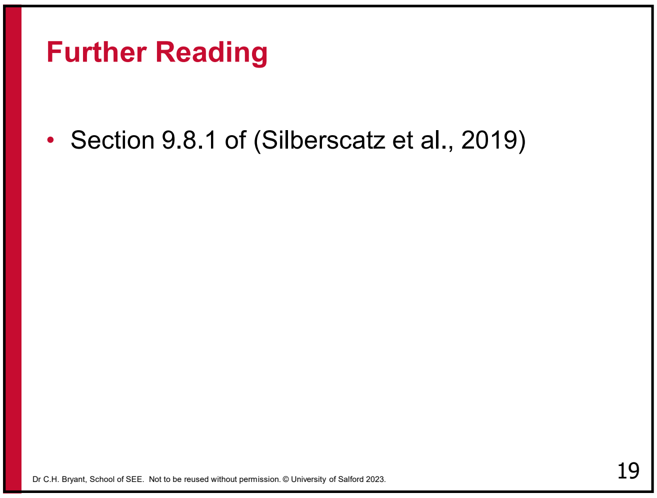


Summary

- SQL injection is a technique whereby a malicious attacker can exploit inadequate data validation to inject arbitrary SQL code into an application's queries and have it executed as though it is a legitimate query.
- An SQL injection attack involves placing SQL statements in the user input, hoping they will reach the database server and be executed alongside the original query.
- The best way to avoid this, is by using **prepared statements** (also referred to as **parameterised queries**).

18

18



Further Reading

- Section 9.8.1 of (Silberscatz et al., 2019)

19